CommCalc

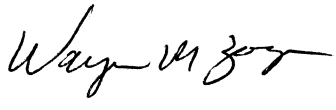
An Honors Thesis (HONRS 499/CS497/CS498)

by

Alexander P. Corn

Thesis Advisor

Wayne M. Zage



Ball State University

Muncie, Indiana

Date: May 8, 2009

Expected Graduation: May 9, 2009

Abstract

The goal of this project was to write a piece of software for calculating optimum commission rates for real estate listings. A realtor inputs information about a potential listing and the computer uses that information, as well as information about the real estate agency's historical listings and business expenses, to extrapolate a commission rate that allows the firm to break even on the listing. This optimum rate can then be used to determine if the listing could be profitable. If the optimum rate is low compared to what the market will bear, the listing is probably safe to take. If the calculated optimum rate is higher than what the market is likely to bear, then the agency should not list the property. Low rates estimated by the system can then be inflated according to the agent's expertise in order to glean more profit from a listing that is sure to sell.

The program also keeps track of all of this information once it is entered into the system. This allows users to evaluate the performance of both the software and the real estate agency over time. Detailed logs are kept in order to assist administrative staff in auditing the information stored in the system. All user accounts are password-protected, ensuring that only authorized employees can access the system.

Artist's Statement

This project doubled as both my computer science software engineering capstone project and my Honors thesis. The software engineering class, CS497 and CS498, spanned two semesters. The premise of the class was to divide into small teams and then design and write a program for a business that had teamed up with the computer science department. My team consisted of Ashley Keith, Jaret Binfond, Wade Guisewhite, and myself. Ashley also used this project as her Honors thesis, and Wade would have as well but he had already started on a different project.

We teamed up with Century 21 because I had done computer maintenance for them in the past and knew that they would be a pleasure to work with. The project started out slowly, largely due to delayed non-disclosure agreement paperwork. Once we had everything we needed, development progressed rapidly. Unfortunately, deadlines always seem to come too soon. While we met all of our client partner's requirements, there were some extra features that we wanted to implement. If we had had the time, we'd have created one feature that graphed different data sets and another that tracked the number of listings sold vs. the number needed to sell to break even.

During the next couple of years, Century 21 Muncie is going to use and evaluate the software we wrote for them. Hopefully the team will be able to provide support and enhancements to it, if we have time. A complete rewrite may also take place. Now that we have done it once, it would be easy to do it over and make better design choices.

Acknowledgements

-I would like to thank my software engineering teammates. Without them, this project would have not been possible.

-I would also like to thank Kerry Wiggerly, our client partner at Century 21 Muncie. This program was his idea and he was instrumental in teaching us how real estate brokerage works as well as how such a business is run. We would have been unable to write any usable code without his insight and direction.

-Finally, I would like to thank Dr. Wayne Zage, our software engineering professor. He led all of the teams throughout the project and ensured that we remained on task throughout the year.

Realty Group Muncie
400 W McGalliard Road
Muncie, Indiana 47303
Business (765) 284-6313
Fax (765) 287-2206
www.thinkrealtygroup.com

Mr. Kerry Wiggerly, President

Century 21 Realty Group Muncie

May 8, 2009

Mr. Wayne Zage, Professor
Ball State University.

Dear Sir,

I am pleased to inform you that Team Burgundy has completed this project in accordance with my expectations. I am confident that the work they accomplished will aid in the operation of my business. It is my hope that this software will continue to serve Century 21 Realty Group Muncie's needs for years to come.

Thank you for this opportunity.

Sincerely,

Kerry Wiggerly

President

2009

Team Burgundy

Jaret Binford, Alex Corn,
Wade Guisewhite, and
Ashley Keith

[USER MANUAL]

[This document details how to use CommCalc. It also briefly explains the algorithm used to compute commission rates. Support details are included at the end.]

GENERAL

Welcome to *CommCalc*! This manual is intended to provide a firm understanding of this application's inner workings, use, and functionality. We will begin in broad strokes and gradually refine to hopefully answer all of your questions.

This manual is not intended to cover every minute detail. Please check the Systems Manual, Javadocs, and the Programming manual for more technical detail.

CommCalc is an application commissioned by Century 21 Muncie. It provides an estimated commission rate for potential property listings. The calculated commission rate is based on historical fixed cost data and allows the user to adjust the parameters used to arrive at the estimate.

User Manual

This manual is intended to provide a detailed description of the program and how to operate it. There are five main sections: description, starting *CommCalc*, using *CommCalc*, error messages, and system recovery. In addition to these sections there are three additional headings to provide adaptability information, frequently asked questions, and contact information. These additional headings should provide a more thorough understanding of certain aspects of the application.

1.2.1 Description

CommCalc has several features and functions. The main function is to estimate a viable commission rate for a potential listing based on historical data, cost analysis, and the properties of the listing in question. *CommCalc* achieves this goal by storing a database of fixed costs as well as accessing the Paragon MLS database. Similar listings and historic fixed costs are used to extrapolate a commission rate. In addition to the optimal commission rate, a maximum commission rate, an average commission rate, a minimum commission rate, and the estimated days on market are also calculated. It is important to note that this application, while theoretically accurate, is limited in the sense that it cannot replace the knowledge of an experienced realtor.

1.2.2 Starting *CommCalc*

To launch *CommCalc* simply double-click the *CommCalc* shortcut on the desktop. The application will launch and you will be greeted with a login screen. On the login screen, you will be prompted for your username and password.

User Manual

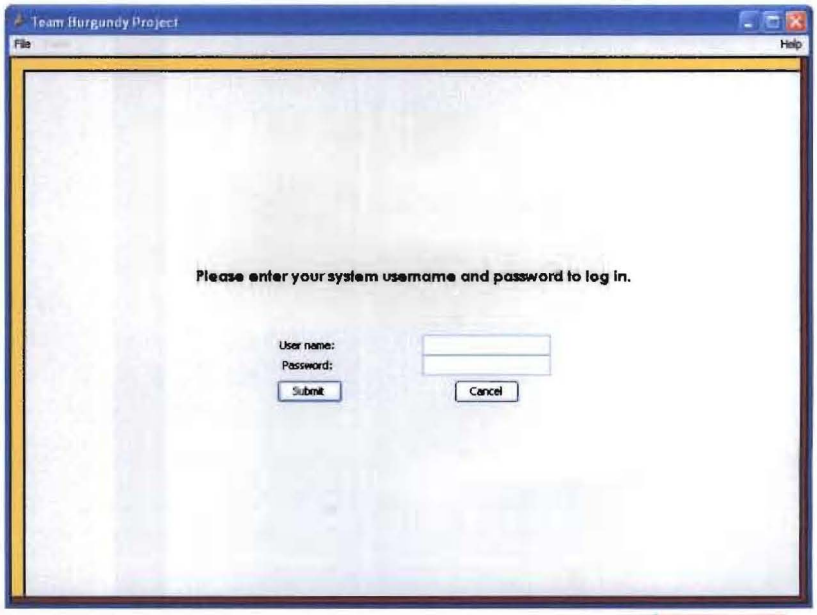
This manual is intended to provide a detailed description of the program and how to operate it. There are five main sections: description, starting *CommCalc*, using *CommCalc*, error messages, and system recovery. In addition to these sections there are three additional headings to provide adaptability information, frequently asked questions, and contact information. These additional headings should provide a more thorough understanding of certain aspects of the application.

1.2.1 Description

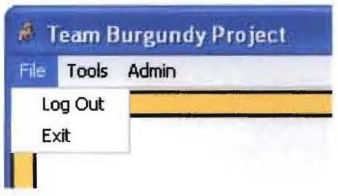
CommCalc has several features and functions. The main function is to estimate a viable commission rate for a potential listing based on historical data, cost analysis, and the properties of the listing in question. *CommCalc* achieves this goal by storing a database of fixed costs as well as accessing the Paragon MLS database. Similar listings and historic fixed costs are used to extrapolate a commission rate. In addition to the optimal commission rate, a maximum commission rate, an average commission rate, a minimum commission rate, and the estimated days on market are also calculated. It is important to note that this application, while theoretically accurate, is limited in the sense that it cannot replace the knowledge of an experienced realtor.

1.2.2 Starting *CommCalc*

To launch *CommCalc* simply double-click the *CommCalc* shortcut on the desktop. The application will launch and you will be greeted with a login screen. On the login screen, you will be prompted for your username and password.



Once you have entered the correct username and password, you will be greeted with a welcome screen. There will be several menu items in the top left of the application window.



File allows the user to log out or exit the application.

Tools grants the user the ability to manage fixed costs, add new fixed cost types, and calculate the commission rate.

Admin allows admin-privileged users the ability to manage users, view user logs, and configure the application. This menu is only available to administrative users. Non-admin users will not be able to see or select this item.

Help shows the user this manual and some information about the application.

1.2.3 Using *CommCalc*

Logging in:

Logging in must be done before the program can be used. Logging in ensures that the system knows who you are. This keeps data safe and it lets the system monitor use of the program.

Step 1: Launch *CommCalc* from your desktop or from the intranet web site.

Step 2: Type your user name in the **User Name** field.

Step 3: Press tab to move your cursor to the **Password** field and type your password.

Step 4: Finally, press **Enter** or click the **Submit** button to log in. You can click the **Cancel** button at any time to close the program instead of logging in.

Step 5: If this is the first time you have logged in, you will be prompted for your Paragon/MLS login credentials. Once they have been entered, they will be stored in the database and you will not be asked for them again unless they change.

Logging out:

Logging out ends your session, but keeps the program running. This allows another user to log in and use the program.

Step 1: Click the top-left **File** menu.

Step 2 Click the **Log Out** menu item to log out. The program will return to the log in screen.

Exit:

Exiting the application stops it from running and frees up resources for other programs to run on the computer.

Step 1: Click the top-left **File** menu.

Step 2 At the bottom of the drop down menu you will see the **Exit** option – click it to exit the application.

Add Cost Type:

*A **Cost Type** defines a category of monthly fixed cost. Once defined, it can be used over and over. Enter a description, such as "Mortgage" or "Utilities." Then you can choose that item when entering monthly fixed costs into the system as described in the next section.*

Step 1: Click the **Tools** menu. Then click the **Add Fixed Cost** option in the **Fixed Cost** submenu.

Step 2: A screen will open showing a **Type** field and a **Submit** button.

Step 3: Enter the new fixed cost type in the **Type** field.

Step 4: Finally, click the **Submit** button at the bottom of the screen.

Add Cost:

This feature associates an expense or income with a month and year. The system uses these records to calculate what monthly costs will be in the future so that a more accurate commission rate can be calculated.

Step 1: Click the **Tools** menu. Scroll down to the **Fixed Cost** submenu and then click **Manage Fixed Costs**.

Step 2: A screen will appear with a scrollable, sortable list of all current fixed costs.

Step 3: Click the **Add** button. A dialog box with four fields will appear.

Step 4: The first field is the **Type** list. It contains all **Fixed Cost** types that have been added to the system as detailed above. Select the particular **Fixed Cost** to add from the drop-down box.

Step 5: Enter the **Amount** of this record in the next field.

Step 6: The third and fourth fields are **Month** and **Year**. These fields specify when the fixed cost you are adding to the system applies. You cannot create a record for the same **Type** more than once in a given month and year. To modify a fixed cost, see the next section titled **Edit Cost.**

Step 7: Choose whether this record is an **Expense** or an **Income**.

Step 8: To commit this record to the database, click the **Add** button beneath the fields. If you decide not to commit this record, click **Cancel**.

Edit Cost:

*Editing a cost is useful if the amount of a cost changes or if a mistake was made when the cost was initially entered. A cost cannot be edited to have the same **Type**, **Month**, and **Year** as another cost.*

Step 1: Click **Tools** and then select the **Fixed Cost** submenu. From there, click on the **Manage Fixed Cost** option.

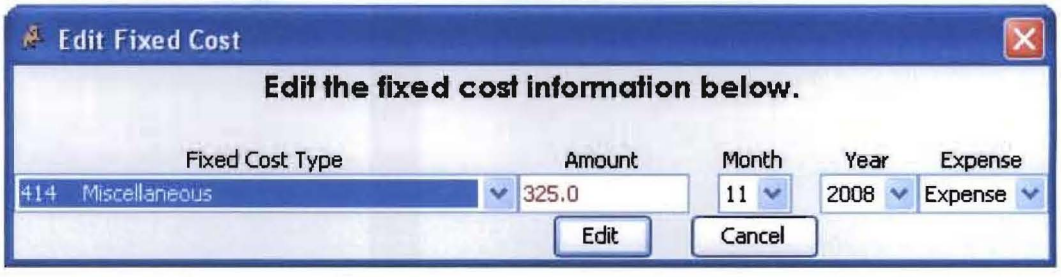
Step 2: A screen will appear with a scrollable, sortable list of all current fixed costs. You will also see, three button options below the list window: **Add**, **Edit**, and **Delete**.

Step 3: Scroll through the list of all current fixed cost and find the record that you would like to edit. Click on it to select and highlight it.

Step 4: With the field selected, click the **Edit** button located below the list of all current fixed costs.

Step 5: A dialog box will appear with the selected fixed cost's information pre-populated in each input field.

Step 6: With the changes made, select the **Edit** button located in the dialog box to commit the changes. To abort the modification, click **Cancel**.



Delete Cost:

Deleting a cost removes it from the database and prevents it from being used in the process of calculating a commission rate.

Step 1: Click the **Tools** menu. Then click the **Manage Fixed Cost** option in the **Fixed Cost** submenu.

Step 2 Scroll through the list of all current fixed costs and find the record you would like to delete. Once you have found it, click it to highlight it.

Step 3 Click the **Delete** button.

Calculate Commission Rate:

This is the main feature of CommCalc. *First it accepts criteria for a potential listing. Next, it queries the MLS system for similar listings. It looks at data from the similar listings, such as selling price, asking price, and days on market. Then it calculates a "break even" commission rate for the potential listing based on the MLS data and on the organization's historical financial data.*

Step 1: Select the **Tools** menu. Then click the **Calculate Commission Rate** option.

Step 2 A screen will appear with several fields of search criteria sorted into four columns: **Use**, **Potential Listing**, **Minimum Search Value**, and **Maximum Search Value**. The **Use** column contains a check box in each row that will determine if the application should include the criteria that the row represents during the search and calculation. The **Potential Listing** column contains all the data for the listing that the user intends to find a commission rate for. The final two columns, **Minimum Search Value** and **Maximum Search Value**, will be populated where there is a range of values to search. These two columns can be manually adjusted if the suggested values are not satisfactory.

Step 3 Fill in all fields that you feel are relevant to this particular listing. You may enable any of the other search fields by checking the **Use** box located next to the value you would like to include in the calculation. It should be noted that the application will only use *exact matches* for the drop-down boxes to calculate the commission rate. Using too many exact-match fields could cause the program to operate unreliably.

Step 4 Finally, click **Calculate** at the bottom of the screen. The computer will query the databases and analyze the results to calculate a commission rate and other information about the listing.

Configure:

This feature allows the administrative user to change the default behavior and threshold parameters of the program. NOTE: The RETS database, mentioned several times in this section, is a computer system that sits behind the normal

MLS website or program. Any data accessed via the MLS system is actually stored in the RETS database. The technical term RETS is used here, but most users will probably be more familiar with the term MLS.

Step 1: Click the **Admin** menu. Then click **Configure**. The **Admin** menu will not be available unless your account has administrative privileges. A screen will appear with a list of configuration options for the program.

RETS URL: This field should be set to the URL of the RETS database. Contact the administrator of that service to obtain this information.

Minimum Listing Records Threshold: If fewer than this many listings match the criteria bounds set by the user for a potential listing in the **Calculate Commission Rate** screen, a warning will be displayed.

Previous Months of Fixed Cost Data Displayed: This field determines how far back fixed cost data can be viewed and modified in the **Manage Fixed Cost** screen.

Commission Rate Percentage Threshold: These values determine what constitutes an outlier in the RETS database based on percentage. Listings with a commission rate below the minimum or above the maximum will not be considered when computing a commission rate.

Commission Rate Dollar Amount Threshold: These values determine what constitutes an outlier in the RETS database based on dollar amounts. Listings with a commission rate below the minimum or above the maximum will not be considered when computing a commission rate.

Number of Months of RETS Listings to Search: This value determines how far back in time to evaluate listings in the RETS system when calculating a commission rate.

Finalizing: To commit the changes to the system, click the **Update** button at the bottom of the screen.

View Logs:

The system tracks all changes that are made in every part of the program. Only the administrator can view these changes and evaluate user activity within the system.

Step 1: Select the **Admin** menu and then click **View Logs**.

Step 2: A list of all **Created Users** is initially displayed. You will find buttons marked **Created Users, Created Fixed Costs, Updated Users, Updated Fixed Costs**, and **Deleted Fixed Costs.** Clicking any one of these buttons will refresh the pane with the requested information. If at any time you would like to refresh the data shown, simply click the **Refresh** button.

View Reports:

Reports are a tool for tracking how accurately the program performs over time. Reports are saved each time a potential listing is analyzed by the system. If the company decides to take on that listing, its information will be entered into the MLS system. Once it is in the MLS system, the MLS number can be entered into the report for that listing so that this program can automatically track the listing by querying the RETS database.

Step 1: Select the **Tools** menu and click **View Reports**. On the left of the screen will be a table of all reports in the system. On the right will be an area that is populated with report details when a report is selected. At the bottom of the screen are three buttons: **Update Report, Change MLS Number**, and **Print Report.**

Changing or Adding an MLS Number: Reports cannot be updated or tracked until an MLS number is associated with it. If the company takes on a listing that was analyzed by this program, the new MLS number should be associated with the saved report that was created when the listing was evaluated. Select the correct report, and then click the **Change MLS Number** button. A window will appear with a text box. Enter the MLS number; then click **Add**.

Updating a Report: Select the report you wish to update from the table on the left. Then click the **Update Report** button. The program will automatically query the MLS system to get all up-to-date information about this particular listing. If an MLS number has not been associated with the selected report, this operation cannot be performed.

Printing a Report: To print a report, simply highlight the desired report and then click the **Print Report** button at the bottom of the screen. Your operating system's print dialog box will appear. Set any desired options, and then confirm the printing. This option only works if a printer is installed on the computer you are using.

Manage Users:

Each person that has access to the system must be issued a user account. User accounts ensure that only authorized people have access to the system. User passwords can be reset by the administrator in case the employee forgets his or her password. Administrators can also disable user accounts if an employee's job no longer requires access to the program or if an employee leaves the company.

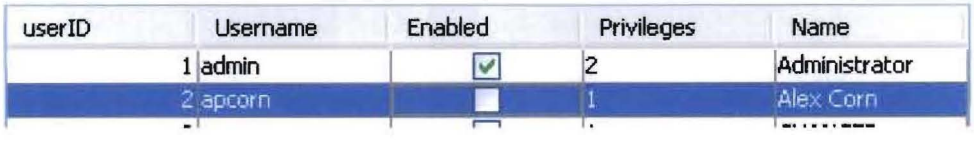
Step 1: Select the **Admin** menu and then click **Manage Users**.

Add: Click the **Add** button to add a user. A window will appear with several fields. Enter a unique username, a password, a privilege level (user or admin), and the user's real name. Click **Add** to add the user, or **Cancel** to close the window without creating a user.

Edit: Highlight an existing user in the list by clicking the row that represents the user account. Then click the **Edit** button. A window will come up with fields pre-populated with the user's existing information. Make any necessary changes, then click **Edit** to commit the changes or **Cancel** to close the window without making the changes.

Set Password: Highlight the user whose password you want to set and then click the **Set Password** button. A window will appear with two fields. Enter the same password twice and then click **Edit** to confirm. The user's password will be changed. You can also click **Cancel** to abort.

Enable/Disable: To enable or disable a user, simply click to toggle the check box in the user account row under the **Enabled** column. The user's ability to log in will be instantly revoked if a checkmark is removed. The ability to log in will be reinstated if a checkmark is replaced.



userID	Username	Enabled	Privileges	Name
1	admin	☑	2	Administrator
2	apcorn	☐	1	Alex Corn

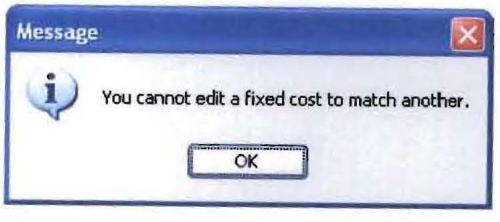
1.2.4 Error Messages

Error checking in *CommCalc* is largely handled by design features in the application. This can be seen by grayed out options in the menu and the current screen. Most fields will not allow you to enter anything but the required type of value. For instance, if the field you are currently typing in expects a numerical value, it will not let you type letters. This is done to prevent passing the program invalid data. It is important for the computer to be able to read all input; any invalid input will result in an unusable calculation.

However, not all errors are preventable. A list of common errors and short explanations follows:

This error happens when the program can't connect to the database. Either the server is down, or your computer isn't connected to the network.

This error indicates that the provided username or password did not match any records in the database. Try again, paying careful attention to spelling.

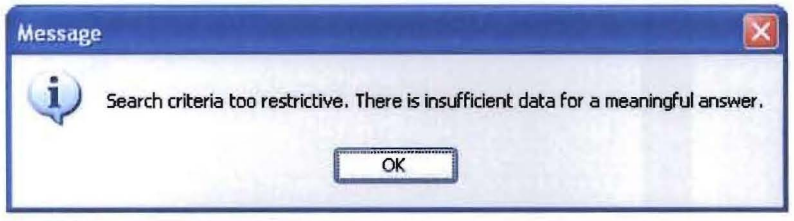


These errors occur when editing Fixed Costs. No two fixed costs can describe the same type, month, and year. If a Fixed Cost is edited or added and those three attributes match an existing record, these errors will be displayed.

This occurs in the Calculate Commission Rate screen. It happens if the program expects a number in one of the text boxes, but the box is either left blank or contains non-numerical characters.

This error indicates that all of the search lines on the Calculate Commission Rate screen have been deselected. Select some search criteria by adding checkmarks along the left side of the screen.

This is another Calculate Commission Rate error. It occurs when the text fields have been selected for a search but have not been filled in with data.



These errors indicate that the values in the Calculate Commission Rate screen are too restrictive. The values do not match enough MLS listings to perform a statistical analysis.

1.2.5 System Recovery

If the application malfunctions, uninstall it from Control Panel (Windows only) and then launch it again from the Web Start. If the problem persists, please contact support.

If the database becomes corrupted, a restore must be performed. Please contact support.

Adaptability

CommCalc has been designed to work on any platform (Windows, Mac, Linux) and on any computer with a Pentium 4 processor and 512MB of RAM or better. Because it is deployed via Web Start, the program only needs to be updated on the server. Once the server is up-to-date, the new version of the program can instantly be run on any workstation configured to run it.

FAQ

Q: How do I install *CommCalc* on my computer?
A:

CommCalc doesn't get "installed" in the traditional sense. It is a program launched from a web server within the company network. Simply browse to the *CommCalc* application page and click the link to launch it. To create a shortcut on your desktop, right-click the link on the web page and select "Copy Shortcut." Then right-click on your desktop and choose "Paste Shortcut."

Q: How do I uninstall *CommCalc*?
A:

To uninstall *CommCalc*, you first need to delete the shortcut from your desktop. You can do this by selecting (single clicking) the icon on your desktop and pressing the delete key. You may also right click the icon and then in the menu select the delete option. Next, go to the Control Panel. Choose the Add or Remove Programs option. Highlight *CommCalc* in the program list and click the Remove button. All the files associated with *CommCalc* will be deleted.

Q: How does *CommCalc*'s math work?
A:

Great question! *CommCalc*'s math is accomplished by a series of calculations which can best be explained by the following eight groupings:

1. Determine when this listing will sell. Query Paragon for similar listings and average the DOM for all closed listings in the result set. Add the average DOM to the current date to get the estimated sale date.

2. Estimate the number of listings that will close in the target month. This is calculated by averaging the number of listings closed during the target calendar month for the last few years. Three years is the default, but this may be adjusted by the administrator.

3. Calculate the average selling price and the standard deviation of listings that closed in the target month for the last few years.

4. Calculate the expected total monthly operating costs of the organization during the target month via extrapolation of fixed cost data in the database.

5. Adjust the total expected monthly operating costs by subtracting the amount of money likely to be made on listings closed by this office but listed by another office. Base this estimate on data from Paragon.

> 5.1. First, find the revenue from such listings (commission rate * sale price) three years and one month ago. (Three years is adjustable by an administrator.) Then find the revenue from such listings from the estimated month of sale three years ago. Find the percent increase between the two values.
>
> 5.2. Repeat the process for two years ago and one year ago.
>
> 5.3. Average all of the percent increases. Then use that percent increase and last month's revenue from closed listings to find the amount of revenue in the estimated month of sale.

6. Adjust the total expected monthly operating costs by subtracting the amount of money likely to be made on properties listed by this office but closed by other offices, similar to Step 5 above.

7. Determine the likelihood of this listing being closed by this office. Find the average number of this office's listings that were closed by this office during the target month over the last few years. Divide that by the average total number of this office's listings that close in that month over the last few years. This includes listings closed by other realty groups.

8. Determine the optimum commission rate.

> 8.1. If the asking price of the listing in question is within one standard deviation of the mean selling price of a listing closed in the target month, then the commission rate * asking price * likelihood the listing will be closed by this office must be equal to or

greater than .68 * total estimated operating costs / number of listings expected to close in the target month.

8.2. If the asking price of the listing in question is more than one standard deviation away from the mean asking price, then the commission rate * asking price * likelihood the listing will be closed by this office must be equal to or greater than .16 * total estimated operating costs / number of listings expected to close in the target month.

Support

If you are experiencing problems with *CommCalc* beyond the cures of this manual, please consult the following list of contacts.

On Site Support:

Alex Corn.....(765) – 425-9061

Troubleshooting:

Jaret Binford.....(317) – 412-1584

Remote Support:

Wade Guisewhite.....wguisewhite@gmail.com

Ashley Keith.....ankeith86@gmail.com

2009

Team Burgundy

Jaret Binford, Alex Corn,
Wade Guisewhite, and
Ashley Keith

[PROGRAMMING MANUAL]

[This document includes an index of all the modules in the source code. Javadocs are provided to describe the appropriate technical use for this program and design decisions are explained.]

Module Index

1.1 . edu.bsu.cs.cs498.teamburgundy

1.1.1 TBProject.java **apcorn, ankeith, waguisewhite**

1.1.1.1 **private void** initCalc( )

1.1.1.2 **public void** calculateCommission(**int** pAskingPrice, **int** lAskingPrice, **int** hAskingPrice, **int** pSQFT, **int** lSQFT, **int** hSQFT, **float** pAcreage, **float** lAcreage, **float** hAcreage, **int** pZip, **int** numBedrooms, **int** numFullBaths, **int** numHalfBaths, **String** township, **String** elem, **String** midd, **String** high, **int** garage1, **int** garage2, **int** foundation, **String** heat, **String** ac)

1.1.1.3 **public void** showLoginScreen( )

1.1.1.4 **public void** showWelcomeScreen( )

1.1.1.5 **public void** showManageFixedCostsScreen( )

1.1.1.6 **public void** showCalculateCommissionRateScreen( )

1.1.1.7 **public void** showAddFixedCostScreen( )

1.1.1.8 **public void** manageUsers( )

1.1.1.9 **public void** about( )

1.1.1.10 **public void** configure( )

1.1.1.11 **public void** showViewReportScreen(**int** i)

1.1.1.12 **public void** showViewLogScreen(**int** i)

1.1.1.13 **public void** gettingStarted( )

1.1.1.14 **public void** manual( )

1.1.1.15 **private void** initDatabase( )

1.1.1.16 **public LinkedList<Object[ ]>** getUsers( )

1.1.1.17 **public LinkedList<Object[ ]>** getLogs(**int** choice)

1.1.1.18 **public LinkedList<String>** getFixedCostTypes( )

1.1.1.19 **public LinkedList<Object[ ]>** getFixedCosts(**int** numMonths)

1.1.1.20 **public boolean** editFixedCost(**int** id, **String** description, **float** amount, **int** month, **int** year, **String** expense)

1.1.1.21 **public boolean** deleteFixedCost(**int** id)

1.1.1.22 **public void** setRETSCredentials(**String** retsUsername, **String** retsPassword, **String** username)

1.1.1.23 **public boolean** createUser(**String** username, **String** password, **int** privilegeLevel, **String** name)

1.1.1.24 **public boolean** editUser(**int** id, **String** username, **int** privilegeLevel, **String** name)

1.1.1.25 **public boolean** enableUser(**int** userID, **boolean** b)

1.1.1.26 **public boolean** setUserPassword(**int** id, **String** password)

1.1.1.27 **public void** addNewFixedCost(**String** description, **float** amount, **int** month, **int** year, **String** expense)

1.1.1.28 **public boolean** addFixedCostType(**String** description)

1.1.1.29 **public Hashtable<CV,Object>** getConfigValues( )

1.1.1.30 **public boolean** setConfigValues(**String** retsURL, **int** numMonthsFixedCostDataDisplayed, **double** commRatePercentHigh, **double** commRatePercentLow, **int** commRateValueHigh, **int** commRateValueLow, **int** numMonthsOfRetsListings, **int** numMonthsReportForecasting)

1.1.1.31 **private void** initAuthenticator( )

1.1.1.32 **public void** logOut( )

1.1.1.33 **public int** authenticateUser(**String** username, **char [ ]** password)
1.1.1.34 **public String** getRealName( )
1.1.1.35 **public** TBProject( )
1.1.1.36 **private void** createAndShowGUI( )
1.1.1.37 **public void** exit( )
1.1.1.38 **public float** getMonthlyCost(**int** month, **int** year, **boolean** expense)
1.1.1.39 **public float** getMonthlyREB(**int** month, **int** year)
1.1.1.40 **public LinkedList<Object [ ]>** getReports( )
1.1.1.41 **public void** createReport(**float** minCom, **float** avgCom, **float** maxCom, **float** estCom, **int** estDOM)
1.1.1.42 **public void** updateReport(**int** reportID, **int** mlsNum)
1.1.1.43 **public Boolean** setReportMLSNum(**int** reportID, **int** mlsNum)

1.1.2 TeamBurgundy.java **apcorn**
1.1.2.1 **public static void** main(**String[ ]** args)
1.1.2.2 **public static void** runProject( )

1.2 . edu.bsu.cs.cs498.teamburgundy.data

1.2.1 CV.java **apcorn**
1.2.2 ElementarySchool.java **ankeith**
1.2.2.1 **private** ElementarySchool(**String** elem, **String** code)
1.2.2.2 **public String** elemString( )
1.2.2.3 **public String** getCode( )
1.2.3 FixedCostTableModel.java **apcorn**
1.2.3.1 **public** FixedCostTableModel(**TBProject** a)
1.2.3.2 **public Object[ ]** getRow(**int** row)
1.2.3.3 **public String** getColumnName(**int** column)
1.2.3.4 **public int** getColumnCount( )
1.2.3.5 **public int** getRowCount( )
1.2.3.6 **public Object** getValueAt(**int** rowIndex, **int** columnIndex)
1.2.3.7 **public void** update( )
1.2.3.8 **public void** months(**int** m)
1.2.4 Foundation.java **ankeith**
1.2.4.1 **private** Foundation(**String** foundation, **int** code)
1.2.4.2 **public String** foundationString( )
1.2.4.3 **public int** getCode( )
1.2.5 Garage.java **ankeith**
1.2.5.1 **private** Garage(**String** garage, **int** code)
1.2.5.2 **public String** garageString( )
1.2.5.3 **public int** getCode( )
1.2.6 HighSchool.java **ankeith**
1.2.6.1 **private** HighSchool(**String** high, **String** code)
1.2.6.2 **public String** highString( )
1.2.6.3 **public String** getCode( )
1.2.7 LogTableModel.java **apcorn**
1.2.7.1 **public** LogTableModel(**TBProject** a)
1.2.7.2 **public Object[ ]** getRow(**int** row)
1.2.7.3 **public void** setChoice(**int** i)
1.2.7.4 **public String** getColumnName(**int** column)
1.2.7.5 **public int** getColumnCount( )
1.2.7.6 **public Class<? extends Object>** getColumnClass(**int** c)
1.2.7.7 **public boolean** isCellEditable(**int** row, **int** col)
1.2.7.8 **public void** setValueAt(**Object** value, **int** row, **int** col)
1.2.7.9 **public int** getRowCount( )
1.2.7.10 **public Object** getValueAt(**int** rowIndex, **int** columnIndex)
1.2.7.11 **public void** setEnabled(**int** row, **boolean** b)
1.2.7.12 **public void** update( )
1.2.8 MiddleSchool.java **ankeith**
1.2.8.1 **private** MiddleSchool(**String** midd, **String** code)
1.2.8.2 **public String** middString( )

1.2.8.3 **public String** getCode( )

1.2.9 ReportTableModel.java **apcorn**

1.2.9.1 **public** ReportTableModel(**TBProject** a)
1.2.9.2 **public Object** [ ] getRow(**int** row)
1.2.9.3 **public int** getColumnCount( )
1.2.9.4 **public Class<? Extends Object>** getColumnClass(**int** c)
1.2.9.5 **public boolean** isCellEditable(**int** row, **int** col)
1.2.9.6 **public void** setValueAt(**Object** value, **int** row, **int** col)
1.2.9.7 **public String** getColumnName(**int** column)
1.2.9.8 **public int** getRowCount( )
1.2.9.9 **public Object** getValueAt(**int** rowIndex, **int** columnIndex)
1.2.9.10 **public void** update( )

1.2.10 RF.java **apcorn, ankeith**

1.2.11 TBUser.java **apcorn, ankeith**

1.2.11.1 **public** TBUser(**int** id, **String** realName, **String** username, **String** retsUsername, **String** retsPassword, **boolean** isAdmin, **TBProject** a)
1.2.11.2 **public boolean** retsConnect( )
1.2.11.3 **public RETSUserSession** getSession( )
1.2.11.4 **public String** getRealName( )
1.2.11.5 **public int** getID( )
1.2.11.6 **public boolean** isAdmin( )

1.2.12 Township.java **ankeith**

1.2.12.1 **private** Township(**String** township, **String** code)
1.2.12.2 **public String** townshipString( )
1.2.12.3 **public String** getCode( )

1.2.13 UserTableModel.java **apcorn**

1.2.13.1 **public** UserTableModel(**TBProject** a)
1.2.13.2 **public Object[ ]** getRow(**int** row)
1.2.13.3 **public String** getColumnName(**int** column)
1.2.13.4 **public int** getColumnCount( )
1.2.13.5 **public Class<? extends Object>** getColumnClass(**int** c)
1.2.13.6 **public boolean** isCellEditable(**int** row, **int** col)
1.2.13.7 **public void** setValueAt(**Object** value, **int** row, **int** col)
1.2.13.8 **public int** getRowCount( )
1.2.13.9 **public Object** getValueAt(**int** rowIndex, **int** columnIndex)
1.2.13.10 **public void** setEnabled(**int** row, **boolean** b)
1.2.13.11 **public void** update( )

1.3 edu.bsu.cs.cs498.teamburgundy.database

1.3.1 Database.java **apcorn, ankeith, waguisewhite**

1.3.1.1 **public** Database( )
1.3.1.2 **public LinkedList<Object[ ]>** getUserCreationLogs( )
1.3.1.3 **public LinkedList<Object[ ]>** getUserUpdateLogs( )
1.3.1.4 **public LinkedList<Object[ ]>** getFixedCostCreationLogs( )
1.3.1.5 **public LinkedList<Object[ ]>** getFixedCostUpdateLogs( )
1.3.1.6 **public LinkedList<Object[ ]>** getFixedCostDeletionLogs( )
1.3.1.7 **public void** setUserID(**int** i)
1.3.1.8 **public boolean** createUser(**String** username, **String** password, **int** privilegeLevel, **String** name)
1.3.1.9 **public boolean** editUser(**int** id, **String** username, **int** privilegeLevel, **String** name)
1.3.1.10 **public boolean** enableUser(**int** id, **boolean** b)
1.3.1.11 **public boolean** setUserPassword(**int** id, **String** password)
1.3.1.12 **public LinkedList<Object[ ]>** getUsers( )
1.3.1.13 **public void** setRETSCredentials(**String** retsUsername, **String** retsPassword, **String** username)
1.3.1.14 **public ResultSet** getAuthInfo(**String** username)
1.3.1.15 **public boolean** deleteFixedCost(**int** id)
1.3.1.16 **public boolean** editFixedCost(**int** id, **String** description, **float** amount, **int** month, **int** year, **String** expense)
1.3.1.17 **public LinkedList<Object[ ]>** getFixedCosts(**int** numMonths)
1.3.1.18 **public LinkedList<Object[ ]>** getDeletedFixedCosts( )

1.3.1.19 **public boolean** addNewFixedCost(**String** description, **float** amount, **int** month, **int** year, **String** expense)
1.3.1.20 **public LinkedList<String>** getFixedCostTypes( )
1.3.1.21 **public boolean** addNewFixedCostType(**String** description)
1.3.1.22 **public boolean** setConfigValues(**String** retsURL, **int** numMonthsFixedCostDataDisplayed,
double commRatePercentHigh, **double** commRatePercentLow, **int** commRateValueHigh,
int commRateValueLow, **int** numMonthsOfRetsListings, **int** numMonthsReportForecasting)
1.3.1.23 **public Hashtable<CV,Object>** getConfigValues( )
1.3.1.24 **public float** getMonthlyIncome(**int** month, **int** year)
1.3.1.25 **public float** getMonthlyExpenses(**int** month, **int** year)
1.3.1.26 **public float** getMonthlyREB(**int** month, **int** year)
1.3.1.27 **public LinkedList<Object[ ]>** getReports( )
1.3.1.28 **public boolean** setReportMLSNum(**int** reportID, **int** mlsNum)
1.3.1.29 **public void** updateReport(**int** reportID, **String** listDate, **String** closeDate, **float** actComRate)
1.3.1.30 **public void** createReport(**float** minCom, **float** avgCom, **float** maxCom, **float** estCom, **int** estDOM)

1.4 edu.bsu.cs.cs498.teamburgundy.gui

1.4.1 AddFixedCostDialog.java **apcorn, ankeith, waguisewhite, jjbinford**
1.4.1.1 **public** AddFixedCostDialog(**TBProject** a, **JFrame** f, **String** t, **ManageFixedCosts** mfc)
1.4.1.2 **private void** clearFields( )
1.4.1.3 **private class** CancelListener **implements** ActionListener
1.4.1.4 **private class** AddListener **implements** ActionListener
1.4.1.5 **private class** ComboDescriptionListener **implements** FocusListener
1.4.1.6 **private class** DescriptionListener **implements** KeyListener

1.4.2 AddMLSNumDialog.java **apcorn, ankeith, waguisewhite**
1.4.2.1 **public** AddMLSNumDialog(**TBProject** a, **JFrame** f, **String** t, **ViewReports** vr, **int** id, **int** mlsNum)
1.4.2.2 **public** AddMLSNumDialog(**TBProject** a, **JFrame** f, **String** t, **ViewReports** vr, **int** id)
1.4.2.3 **private class** CancelListener **implements** ActionListener
1.4.2.4 **private class** AddListener **implements** ActionListener

1.4.3 AddUserDialog.java **apcorn, ankeith, waguisewhite**
1.4.3.1 **public** AddUserDialog(**TBProject** a, **JFrame** f, **String** t, **ManageUsers** mu)
1.4.3.2 **private class** CancelListener **implements** ActionListener
1.4.3.3 **private class** AddListener **implements** ActionListener

1.4.4 EditFixedCostDialog.java **apcorn, ankeith, waguisewhite**
1.4.4.1 **public** EditFixedCostDialog(**TBProject** a, **JFrame** f, **String** t, **ManageFixedCosts** mfc, **int** id,
Object [ ] fixedCostData)
1.4.4.2 **private class** CancelListener **implements** ActionListener
1.4.4.3 **private class** EditListener **implements** ActionListener

1.4.5 EditUserDialog.java **apcorn, ankeith, waguisewhite**
1.4.5.1 **public** EditUserDialog(**TBProject** a, **JFrame** f, **String** t, **ManageUsers** mu, **int** id, **Object [ ]** userdata)
1.4.5.2 **private class** CancelListener **implements** ActionListener
1.4.5.3 **private class** EditListener **implements** ActionListener

1.4.6 SetUserPasswordDialog.java **apcorn, ankeith, waguisewhite**
1.4.6.1 **public** SetUserPasswordDialog(**TBProject** a, **JFrame** f, **String** t, **String** username, **int** id)
1.4.6.2 **private class** CancelListener **implements** ActionListener
1.4.6.3 **private class** SetPasswordListener **implements** ActionListener

1.4.7 TBFocusTraversalPolicy.java **apcorn**
1.4.7.1 **public** TBFocusTraversalPolicy(**Vector<Component>** order)
1.4.7.2 **public Component** getComponentAfter(**Container** container, **Component** component)
1.4.7.3 **public Component** getComponentBefore(**Container** container, **Component** component)
1.4.7.4 **public Component** getDefaultComponent(**Container** container)
1.4.7.5 **public Component** getFirstComponent(**Container** container)
1.4.7.6 **public Component** getLastComponent(**Container** container)

1.4.8 TBMenu.java **apcorn**
1.4.8.1 **public** TBMenu(**TBProject** a)
1.4.8.2 **public void** enableMenus(**boolean** a)
1.4.8.3 **public void** disableMenus( )
1.4.8.4 **private class** ConfigureListener **implements** ActionListener

1.4.8.5 **private class** ViewReportsListener **implements** ActionListener
1.4.8.6 **private class** ViewLogsListener **implements** ActionListener
1.4.8.7 **private class** ManualListener **implements** ActionListener
1.4.8.8 **private class** AddFixedCostListener **implements** ActionListener
1.4.8.9 **private class** ManageFixedCostsListener **implements** ActionListener
1.4.8.10 **private class** ManageUsersListener **implements** ActionListener
1.4.8.11 **private class** CalculateCommissionRateListener **implements** ActionListener
1.4.8.12 **private class** AboutListener **implements** ActionListener
1.4.8.13 **private class** LogOutListener **implements** ActionListener
1.4.8.14 **private class** ExitListener **implements** ActionListener

1.4.9 TBRETSDialog.java **apcorn**
1.4.9.1 **public** TBRETSDialog(**TBProject** a, **JFrame** f, **String** t, **String** username, **char [ ]** password)
1.4.9.2 **private class** CancelListener **implements** ActionListener
1.4.9.3 **private class** AddListener **implements** ActionListener

1.5 edu.bsu.cs.cs498.teamburgundy.helpers

1.5.1 Authenticator.java **apcorn**
1.5.1.1 **public** Authenticator(**Database** d)
1.5.1.2 **public TBUser** auth(**TBProject** a, **String** username, **char [ ]** password)

1.5.2 Calculate.java **apcorn, ankeith**
1.5.2.1 **public** Calculate(**TBProject** a, **TBUser** u)
1.5.2.2 **public void** findListingsClosed( )
1.5.2.3 **private void** populateListingData( )
1.5.2.4 **public int** extrapolateMini(**int [ ]** target, **int [ ]** last)
1.5.2.5 **public void** extrapolate( )
1.5.2.6 **public void** calculateCommission( )
1.5.2.7 **private boolean** userInputQueryAndCalculations( )
1.5.2.8 **private float** parseCommissionRate(**String** s, **int** askingPrice)
1.5.2.9 **private int** endDayOf(**int** m)
1.5.2.10 **public boolean** buildQuery(**int** pAskingPrice, **int** lAskingPrice, **int** hAskingPrice, **int** pSQFT, **int** lSQFT, **int** hSQFT, **float** pAcreage, **float** lAcreage, **float** hAcreage, **int** pZip, **int** numBedrooms, **int** numFullBaths, **int** numHalfBaths, **String** township, **String** elem, **String** midd, **String** high, **int** garage1, **int** garage2, **int** foundation, **String** heat, **String** ac)
1.5.2.11 **private void** updateConfigValues( )

1.5.3 DateUtil.java **apcorn**
1.5.3.1 **public static Date** nowPlusDays(**long** days)
1.5.3.2 **public static long** daysBetween(**Date** d1, **Date** d2)

1.6 edu.bsu.cs.cs498.teamburgundy.panel

1.6.1 CalculateCommissionRate.java **apcorn, ankeith, waguisewhite**
1.6.1.1 **public** CalculateCommissionRate(**TBProject** app, **Dimension** dimension, **JFrame** f)
1.6.1.2 **public void** initShow( )
1.6.1.3 **private void** getTownshipComboBoxOptions(**JComboBox** b)
1.6.1.4 **private void** getElemComboBoxOptions(**JComboBox** b)
1.6.1.5 **private void** getMiddComboBoxOptions(**JComboBox** b)
1.6.1.6 **private void** getHighComboBoxOptions(**JComboBox** b)
1.6.1.7 **private void** getGarageComboBoxOptions(**JComboBox** b)
1.6.1.8 **private void** getFoundationComboBoxOptions(**JComboBox** b)
1.6.1.9 **public FocusTraversalPolicy** getFocusTraversalPolicy( )
1.6.1.10 **private class** CalculateListener **implements** ActionListener
1.6.1.11 **private class** CheckBoxListener **implements** ItemListener
1.6.1.12 **private class** pAskPriceTextFieldKeyListener **implements** KeyListener
1.6.1.13 **private class** pSQFTTextFieldKeyListener **implements** KeyListener
1.6.1.14 **private class** pAcreageTextFieldKeyListener **implements** KeyListener

1.6.2 Configure.java **apcorn, ankeith, waguisewhite**
1.6.2.1 **public** Configure(**TBProject** a, **Dimension** d, **JFrame** f)
1.6.2.2 **public void** initShow( )
1.6.2.3 **private void** setInitValues( )
1.6.2.4 **private class** ChangeListener **implements** ActionListener

1.6.3 LogIn.java **apcorn, ankeith, waguisewhite**
- 1.6.3.1 **public** LogIn(**TBProject** app, **Dimension** dimension, **JFrame** f)
- 1.6.3.2 **public void** initShow( )
- 1.6.3.3 **private class** CancelListener **implements** ActionListener
- 1.6.3.4 **private class** SubmitListener **implements** ActionListener

1.6.4 ManageFixedCosts.java **apcorn, ankeith, waguisewhite**
- 1.6.4.1 **public** ManageFixedCosts(**TBProject** a, **Dimension** d, **JFrame** f)
- 1.6.4.2 **public void** initShow( )
- 1.6.4.3 **private void** enableButtons(**boolean** b)
- 1.6.4.4 **public void** reloadFixedCostData( )
- 1.6.4.5 **private class** DeleteListener **implements** ActionListener
- 1.6.4.6 **private class** EditListener **implements** ActionListener
- 1.6.4.7 **private class** AddListener **implements** ActionListener
- 1.6.4.8 **private void** updateConfigValues( )
- 1.6.4.9 **private class** FixedCostTableSelectionListener **implements** ListSelectionListener

1.6.5 ManageUsers.java **apcorn, ankeith, waguisewhite**
- 1.6.5.1 **public** ManageUsers(**TBProject** a, **Dimension** d, **JFrame** f)
- 1.6.5.2 **private void** enableButtons(**boolean** b)
- 1.6.5.3 **public void** initShow( )
- 1.6.5.4 **public void** reloadUserData( )
- 1.6.5.5 **private class** SetPWListener **implements** ActionListener
- 1.6.5.6 **private class** EditListener **implements** ActionListener
- 1.6.5.7 **private class** UserTableSelectionListener **implements** ListSelectionListener
- 1.6.5.8 **private class** AddListener **implements** ActionListener

1.6.6 NewFixedCost.java **apcorn, ankeith, waguisewhite**
- 1.6.6.1 **public** NewFixedCost(**TBProject** a, **Dimension** d, **JFrame** f)
- 1.6.6.2 **public void** initShow( )
- 1.6.6.3 **private class** SubmitListener **implements** ActionListener
- 1.6.6.4 **private class** DescriptionListener **implements** KeyListener

1.6.7 ViewLogs.java **apcorn, ankeith, waguisewhite**
- 1.6.7.1 **public** ViewLogs(**TBProject** a, **Dimension** d, **JFrame** f)
- 1.6.7.2 **public void** initShow( )
- 1.6.7.3 **public void** reloadLogData( )
- 1.6.7.4 **private class** UserCreateListener **implements** ActionListener
- 1.6.7.5 **private class** UserUpdateListener **implements** ActionListener
- 1.6.7.6 **private class** FixedCostCreateListener **implements** ActionListener
- 1.6.7.7 **private class** FixedCostUpdateListener **implements** ActionListener
- 1.6.7.8 **private class** FixedCostDeleteListener **implements** ActionListener
- 1.6.7.9 **private class** LogTableSelectionListener **implements** ListSelectionListener

1.6.8 ViewReports.java **apcorn, ankeith, waguisewhite**
- 1.6.8.1 **public** ViewReports(**TBProject** a, **Dimension** d, **JFrame** f)
- 1.6.8.2 **public void** initShow( )
- 1.6.8.3 **public void** reloadReportData( )
- 1.6.8.4 **private void** showReport(**int** reportNum)
- 1.6.8.5 **private void** enableButtons(**boolean** b)
- 1.6.8.6 **private class** PrintReport **implements** Printable, ActionListener
- 1.6.8.7 **private static class** MLSNumRenderer **extends** DefaultTableCellRenderer
- 1.6.8.8 **private class** ReportTableSelectionListener **implements** ListSelectionListener
- 1.6.8.9 **private class** AddListener **implements** ActionListener
- 1.6.8.10 **private class** UpdateListener **implements** ActionListener

1.6.9 WelcomeScreen.java **apcorn, ankeith, waguisewhite**
- 1.6.9.1 **public** WelcomeScreen(**TBProject** app, **Dimension** dimension)
- 1.6.9.2 **public void** inItShow( )

1.7 edu.bsu.cs.cs498.teamburgundy.TBinterface

1.7.1 TBPanel.java **apcorn**
- 1.7.1.1 **public void** initShow( )

Design Decisions

Why use the Web Launch?

Using the Web Launch for the application makes distribution easy. The application is located online and can be accessed from multiple computers in multiple locations. Additionally, using the Web Launch enhances the maintainability of the application. Updates to the software can be posted online once and downloaded by everyone using the program.

Why use Java as the programming language?

Java was a language that several members of the team were already familiar with. Familiarity with a language helps programmers create code of a better quality more efficiently. Furthermore, Java has a convenient tool for documentation. Javadocs are generated automatically from formatted comments in the code. The Javadocs for this program are provided at the end of this manual. They explain the purpose and technical use of the various classes and methods in the code.

Why use enumerated types?

Enumerated types are easy to change. Because our program is also reliant on the RETS databases, which can vary between implementations, we tried to make it simple to make modifications for updates or extensibility.

Why use Array Lists of Linked Lists?

This technique makes the data structure flexible so that the administrator can configure and control the amount of data processed when calculating a commission rate.

Why pass a bunch of parameters instead of an Object when searching for similar listings?

Creating an Object with a bunch of public fields violates principles of object-oriented programming because it exposes implementation.

2009

Team Burgundy

Jaret Binford, Alex Corn,
Wade Guisewhite, and
Ashley Keith

[OPERATION MANUAL]

[This document provides an overview of system operations. The manual includes details on installation, recovery, shutdown, and backup.]

Overview

The platform that *CommCalc* runs on provides numerous advantages in addition to the functions of the software itself. We have selected common, free software to facilitate ease of deployment and minimization of costs. Both qualities are paramount to providing a useful software product. We have also chosen a system that will be easy to maintain and upgrade. Ease of maintenance will allow the developers to work faster and more efficiently, saving money for the client and speeding up delivery times for updates and feature requests.

This application needs to be run on a computer with a Pentium 3 class processor or better. The system is designed to be deployed from a server and requires no local installation. Additionally, the application stores data on the server side which is backed up and independently secure. This means that we are able to correct any application errors server side and quickly fix the problem on all installations.

Subversion, an open-source version control system, was used throughout development and continues to be used when updates are made. This provides a method of recovery because Subversion allows the developers to revert to any previous state, regardless of our "progress."

System Installation

The application is designed to be deployed from a server. This means that no installation to a local machine is necessary. The .jar and .jnlp files must be copied to a web (http) folder accessible on the intranet. A shortcut to the .jnlp file will allow users to launch the application from their desktops.

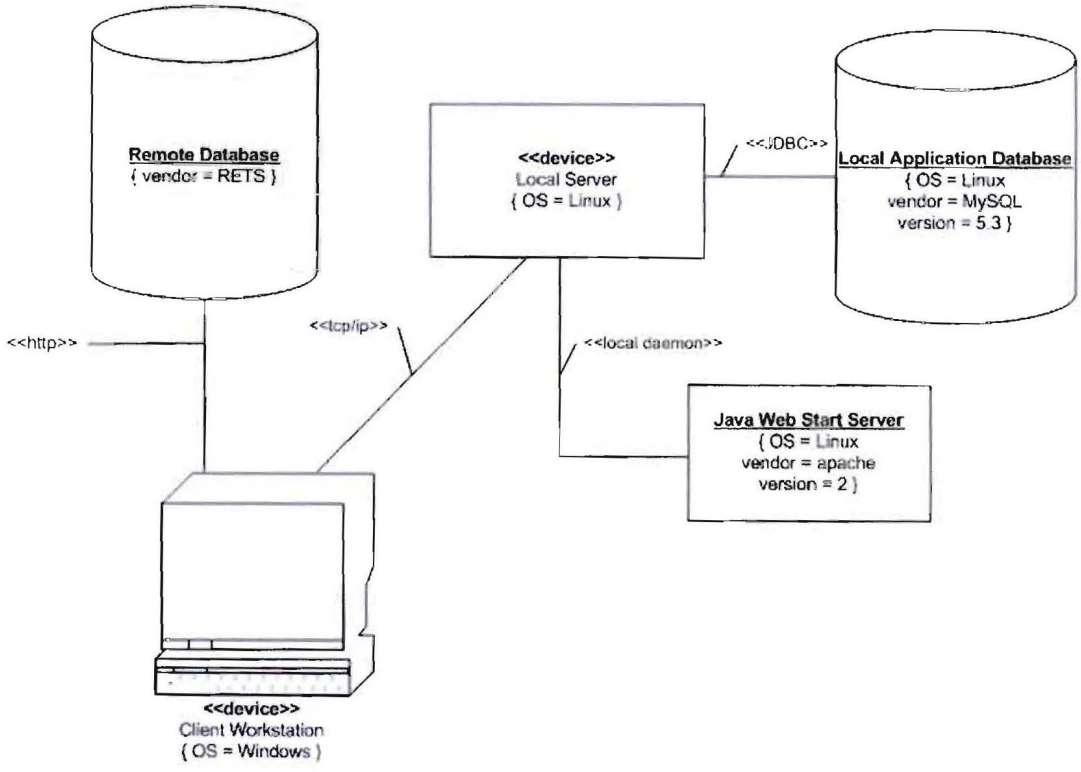
In addition to copying the files to the web server, a MySQL server must also be set up. To create the database, issue the following commands at the MySQL prompt on your server. The commCalc.sql file can be found in the application distribution directory.

```
mysql> CREATE DATABASE name; GRANT ALL ON *.* TO 'username'@'%';

mysql>quit;

bash$ mysql -u username -p password databaseNameFromAboveLines < commCalc.sql
```

Our system deployment can best be understood by the following deployment diagram:



Recovery

The application can be restored to any previous version remotely from the server side. The data is stored on an external server which allows data security. This means that no critical data will be lost upon system failure. However, any changes that were in progress will not be saved. For instance, if a user is adding a new fixed cost and has not submitted by clicking the "Add" button, that data will be lost. This data can easily be reentered once the application has been restored to its previous working condition.

Shutdown

Shutting down the application is handled several ways. The user may close the window of the application by pressing the close button (represented by an "X") located in the top right of the application window. The user may also choose to exit the application from the **File** menu located in the top left of the application window.

Shutting down the system will set all user session values to null. This means that whenever the application is exited, the log off procedure is performed for whatever user is currently logged in using that session. This severs the connection to the RETS database and the application database. Should this process not complete, the user would not be able to log into the application the next time they ran the program.

Backup

Backup of our application is handled completely automatically. All reports, fixed costs, user data, and logs are written to a separate database which is secure on an external server. The rest of the application backup, the RETS database, is handled by RETS and any queries regarding their backup procedure should be submitted to the appropriate RETS contact person. In addition, Subversion is used to save our entire application and all its various states of completion for later use or reversion.

2009

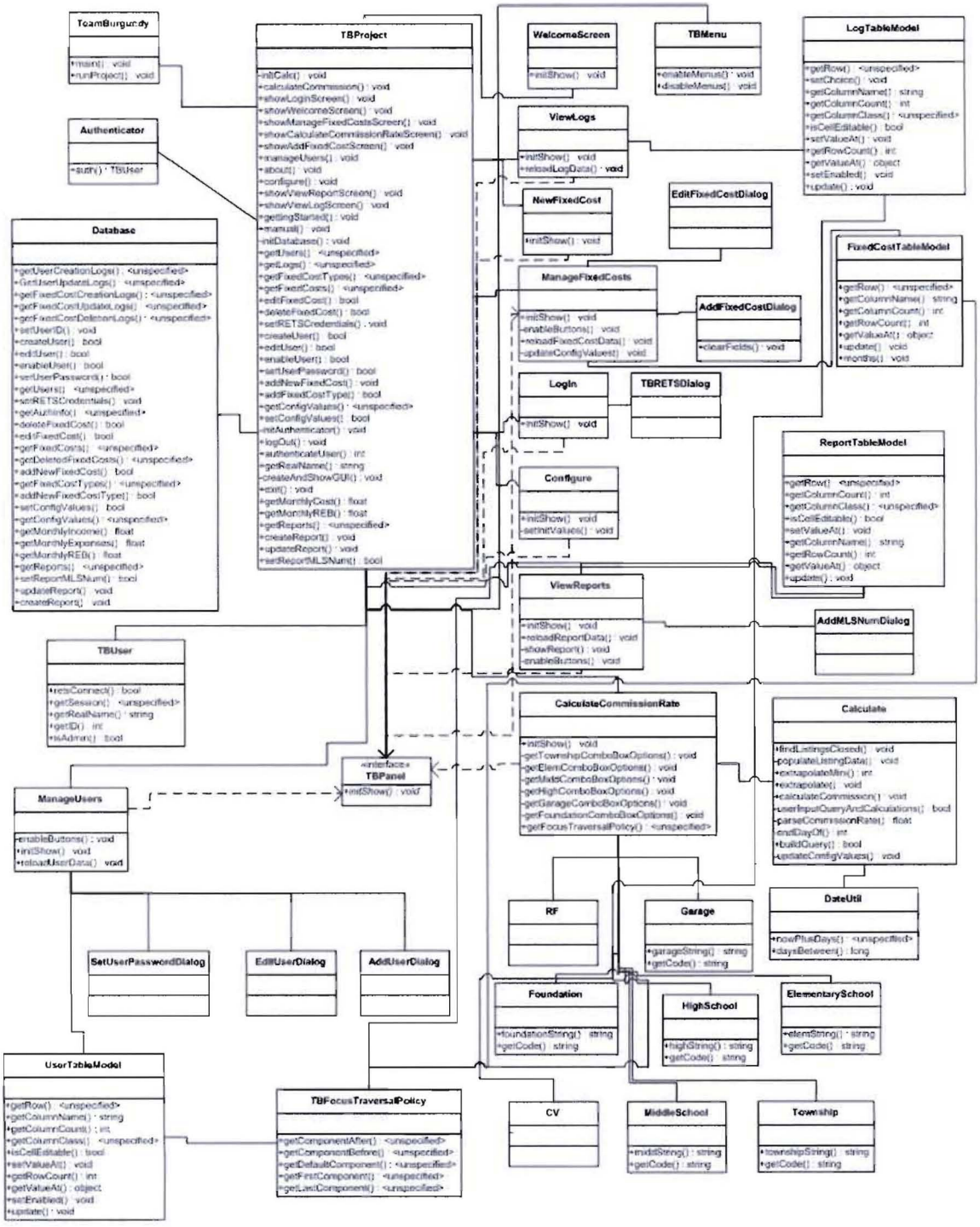
Team Burgundy

Jaret Binford, Alex Corn,
Wade Guisewhite, and
Ashley Keith

[SYSTEMS MANUAL]

[This document describes the design and testing of the system. The manual includes diagrams, graph matrices, and use cases that were used throughout the software engineering process.]

Class Diagram



Test Data

1.2.1 Acceptance Testing

At the end of our project, we scheduled an acceptance test with our client partner. In order to pass our acceptance test, we had to install the application on a workstation at Century 21 Muncie and demonstrate to the owner that the application met his requirements. In addition to this demonstration, we also provided an explanation of our program's mathematical functions.

The following requirements were met and demonstrated:

1. An operations manual with screen shots must be provided to give employees a step-by-step method of using the software product.
2. The new software must work with existing systems and multiple terminals.
3. The program must take into account historical data archived at Century 21.
4. The system must be secure with limited accessibility for:
    a. the administrative team
    b. the sales manager
    c. the general manager
5. Users must have unique identifiers, but they will all have the same permissions.
6. Users will have distinct accounts to facilitate activity logging.
7. An update feature for fixed costs must be provided.
8. The program must allow for the creation of a new cost to be entered.
9. The program should allow for anticipated, one-time costs.
10. The system should create reports as files that can be archived to disk or printed.
11. A comprehensive search subsystem must be present.

To explain the mathematics and data mining involved, we presented the following steps:

1. Determine when this listing will sell. Query Paragon for similar listings and average the DOM for all closed listings in the result set. Add the average DOM to the current date to get the estimated sale date.

2. Estimate the number of listings that will close in the target month. This is calculated by averaging the number of listings closed during the target calendar month for the last few years. Three years is the default, but this may be adjusted by the administrator.

3. Calculate the average selling price and the standard deviation of listings that closed in the target month for the last few years.

4. Calculate the expected total monthly operating costs of the organization during the target month via extrapolation of fixed cost data in the database.

5. Adjust the total expected monthly operating costs by subtracting the amount of money likely to be made on listings closed by this office but listed by another office. Base this estimate on data from Paragon.

5.1. First, find the revenue from such listings (commission rate * sale price) three years and one month ago. (Three years is adjustable by an administrator.) Then find the revenue from such listings from the estimated month of sale three years ago. Find the percent increase between the two values.

5.2. Repeat the process for two years ago and one year ago.

5.3. Average all of the percent increases. Then use that percent increase and last month's revenue from closed listings to find the amount of revenue in the estimated month of sale.

6. Adjust the total expected monthly operating costs by subtracting the amount of money likely to be made on properties listed by this office but closed by other offices, similar to Step 5 above.

7. Determine the likelihood of this listing being closed by this office. Find the average number of this office's listings that were closed by this office during the target month over the last few years. Divide that by the average total number of this office's listings that close in that month over the last few years. This includes listings closed by other realty groups.

8. Determine the optimum commission rate.

8.1. If the asking price of the listing in question is within one standard deviation of the mean selling price of a listing closed in the target month, then the commission rate * asking price * likelihood the listing will be closed by this office must be equal to or greater than .68 * total estimated operating costs / number of listings expected to close in the target month.

8.2. If the asking price of the listing in question is more than one standard deviation away from the mean asking price, then the commission rate * asking price * likelihood the listing will be closed by this office must be equal to or greater than .16 * total estimated operating costs / number of listings expected to close in the target month.

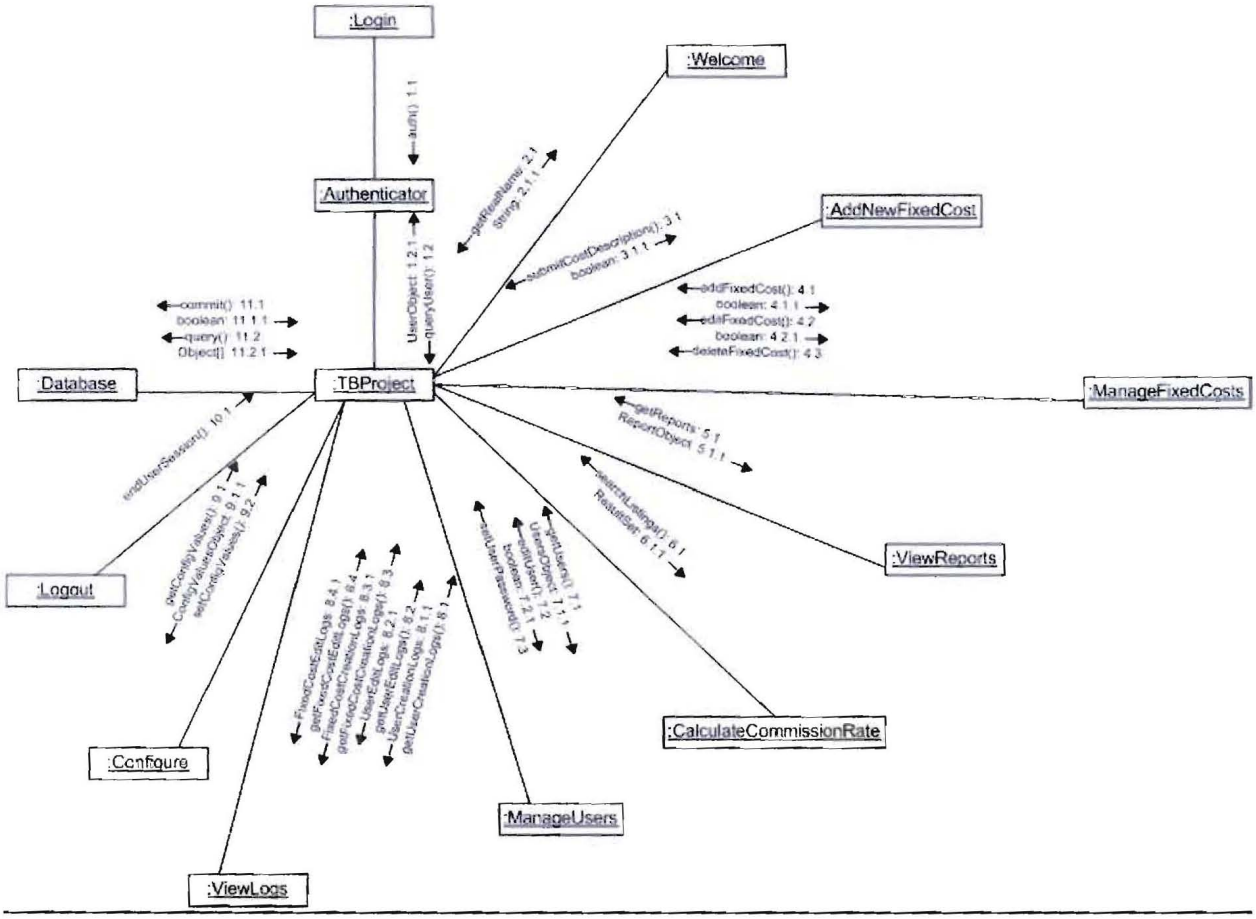
1.2.2 Path Testing

To ensure that our program is free from as many defects as possible we have chosen to demonstrate that our application has complete path coverage, complete condition coverage, and expected functionality. In order to show this we chose to use our class diagram in correlation with graph matrices (see *Section 1.2.4*) to prove path coverage. This had the added benefit of providing condition coverage.

In order to ensure that the program met all functional requirements, we used each of our use cases as a test case. While this did not showcase absolute coverage, it provided a nearly exhaustive run of random black box-style testing. The use cases also helped test the systems design and function intent.

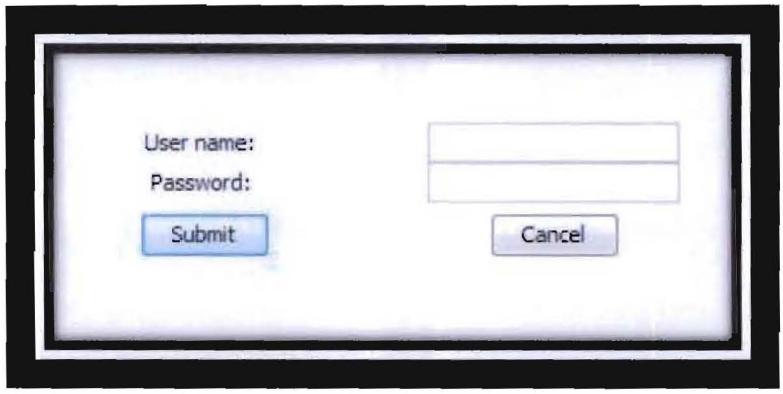
Finally, by using our collaboration diagram, we were able to stress relationships and find any weaknesses that might have occurred in those vital sections of code.

1.2.3 Collaboration Testing

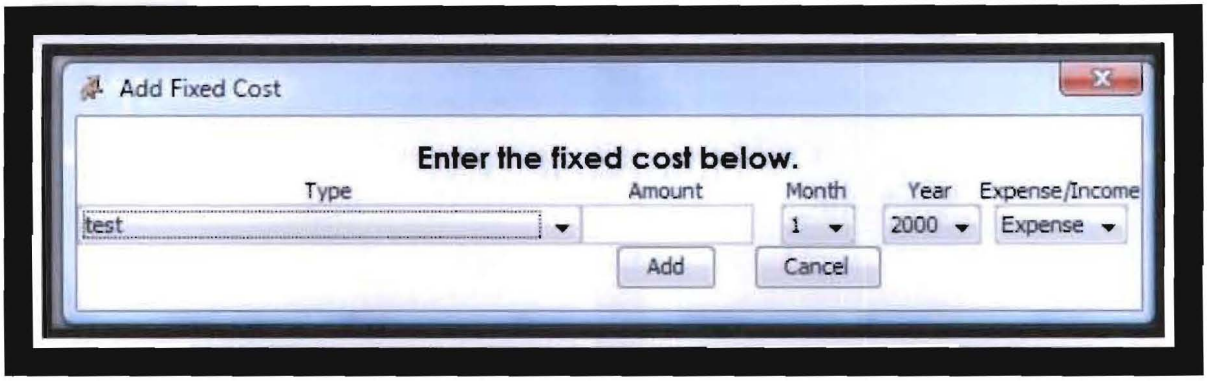


By creating test cases that focused on the relationships between each of these key functions we were able to test aspects of the application not directly accessible from the GUI. Most importantly, we tested the connections between Database and TBProject, Authenticator and TBProject, TBProject and UserManagement, and TBProject and CalculateCommissionRate. The collaboration testing was accomplished by submitting queries to the database for these specific tasks and functions, ensuring that at least one test of each case occurred and that all completed successfully.

1.2.4 Graph Matrices

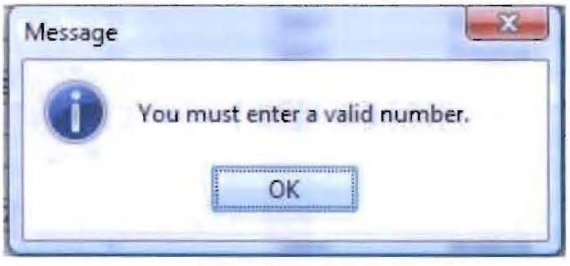


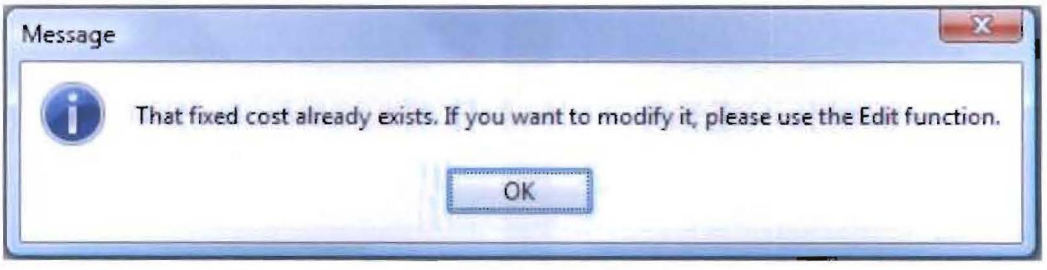
CASE 1	CASE 2	CASE 3
Valid Username	**Invalid Username**	Valid Username
Valid Password	Valid Password	**Invalid Password**



CASE 1	CASE 2	CASE 3
Valid Amount	**Invalid Amount**	Valid Amount
Valid Type/Month/Year/E&I	Valid Type/Month/Year/E&I	**Invalid Type/Month/Year/E&I**
Valid Function call add/cancel	Valid Function call add/cancel	Valid Function call add/cancel

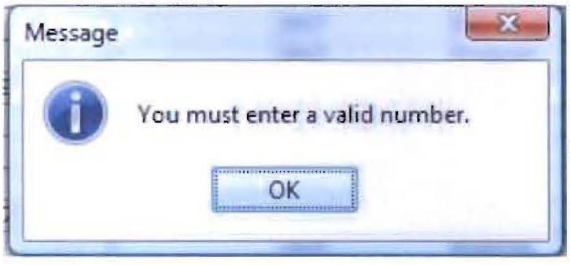
An invalid amount consists of a non-float, non-integer value, including leaving the field blank. An invalid type/month/year/E&I occurs when the type, month, year, and E&I match an existing record. An invalid function call of add/cancel would indicate that the user has not selected an option and is simply not interacting with the Add Fixed Cost dialog box.

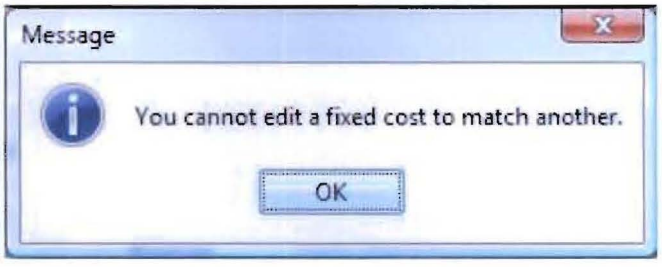


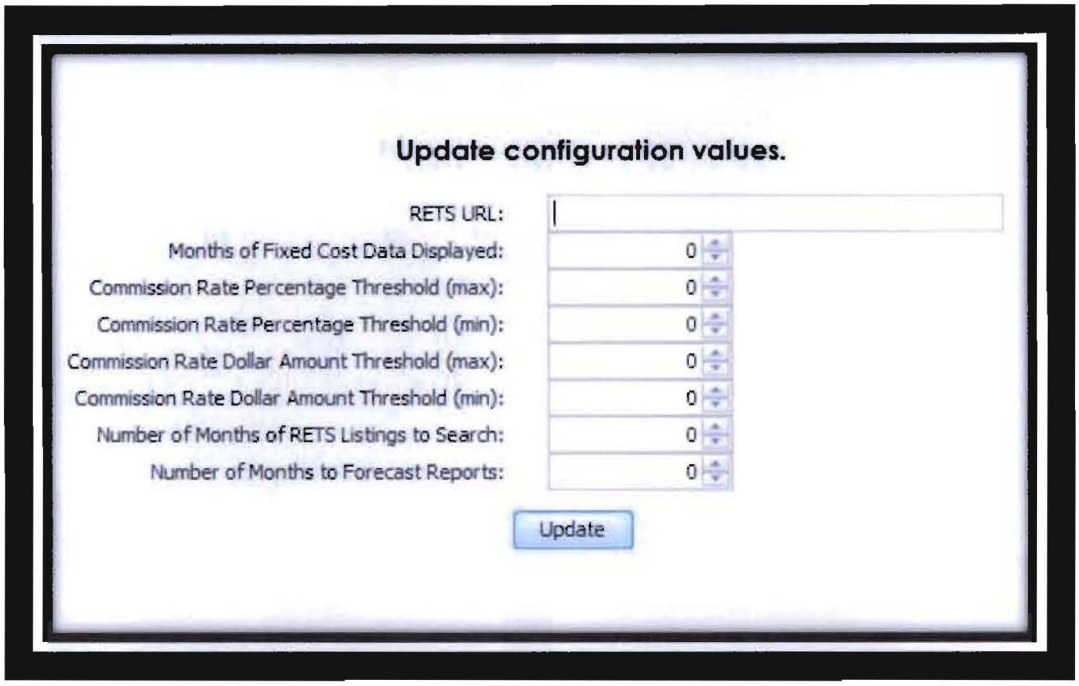


CASE 1	CASE 2	CASE 3
Valid Amount	**Invalid Amount**	Valid Amount
Valid Type/Month/Year/E&I	Valid Type/Month/Year/E&I	**Invalid Type/Month/Year/E&I**
Valid Function call add/cancel	Valid Function call add/cancel	Valid Function call add/cancel

An error can occur if you update the amount to a non-float, non-integer value, including leaving the field blank. An invalid type/month/year/E&I occurs when the type, month, year, and E&I match an existing record. An invalid function call of add/cancel would indicate that the user has not selected an option and is simply sitting at the Edit Fixed Cost dialog box.



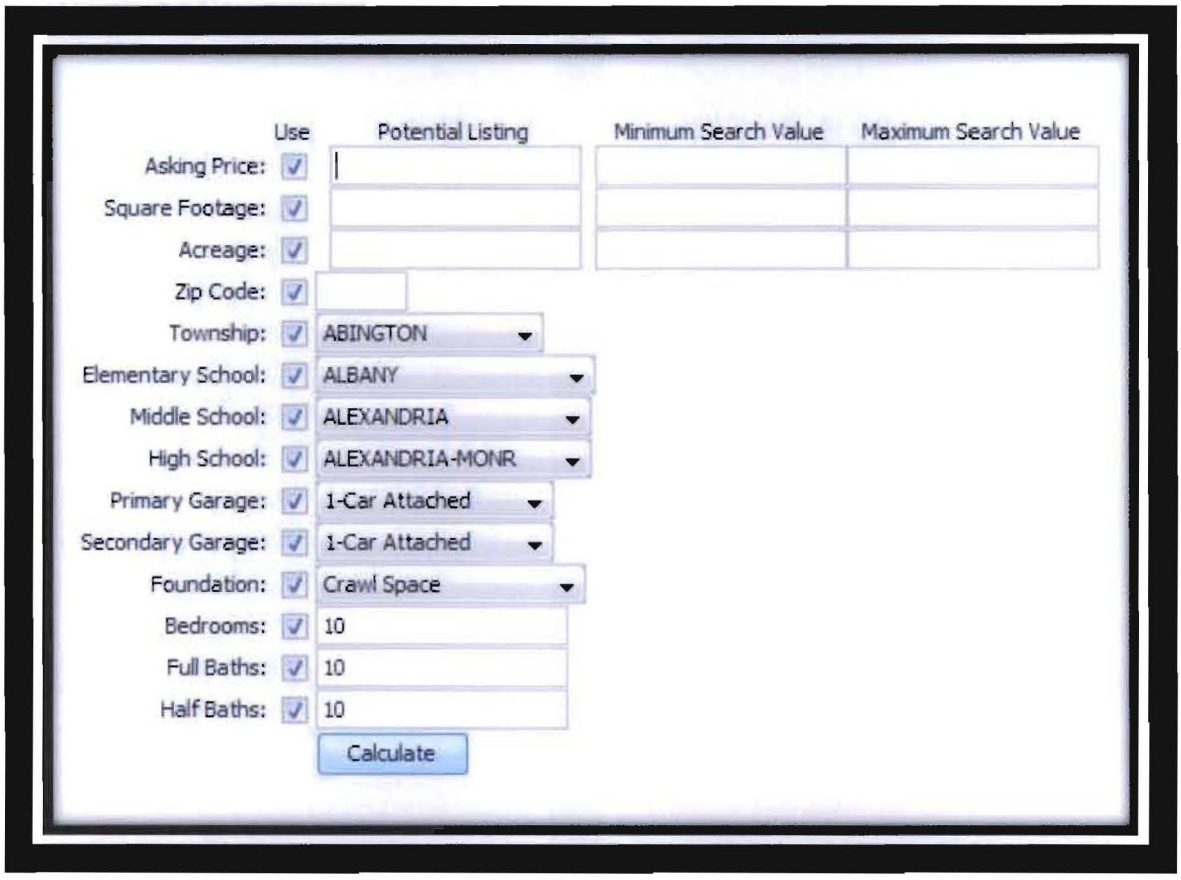




	1	2	3	4	5	6	7	8	9
RETS	V	I	v	v	v	v	v	v	v
MCost	V	v	I	v	v	v	v	v	v
CommPM	V	v	v	I	v	v	v	v	v
mCommP	V	v	v	v	I	v	v	v	v
CommDM	V	v	v	v	v	I	v	v	v
mCommD	V	v	v	v	v	v	I	v	v
#Mrets	V	v	v	v	v	v	v	I	v
#MFore	V	v	v	v	v	v	v	v	I

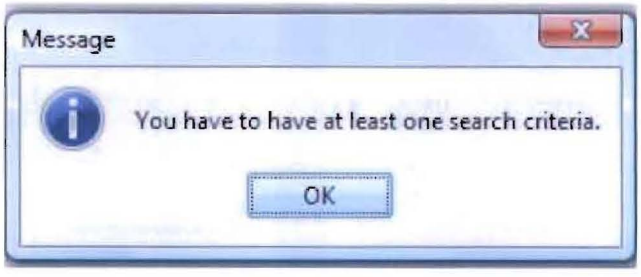
RETS:	The RETS URL field
Mcost :	Months of Fixed Cost Data Displayed field
CommPM:	Commission Rate Percentage Threshold (max) field
mCommP:	Commission Rate Percentage Threshold (min) field
CommDM:	Commission Rate Dollar Amount Threshold (max) field
mCommDM:	Commission Rate Dollar Amount Threshold (min) field
#Mrets:	Number of Months of RETS Listings to Search field
#Mfore:	Number of Months to Forecast Reports field

Besides the initial valid test case, each test case above checks an invalid field within the configuration menu. For instance, if you change the mCommPM to a non-integer, non-float value you will receive an error notifying you that the system refused the input. Typing a character in the mCommPM field will not be accepted by the program either. In the case of the RETS URL, should the user enter an invalid string, the application will recognize the malformed data and deny the change.



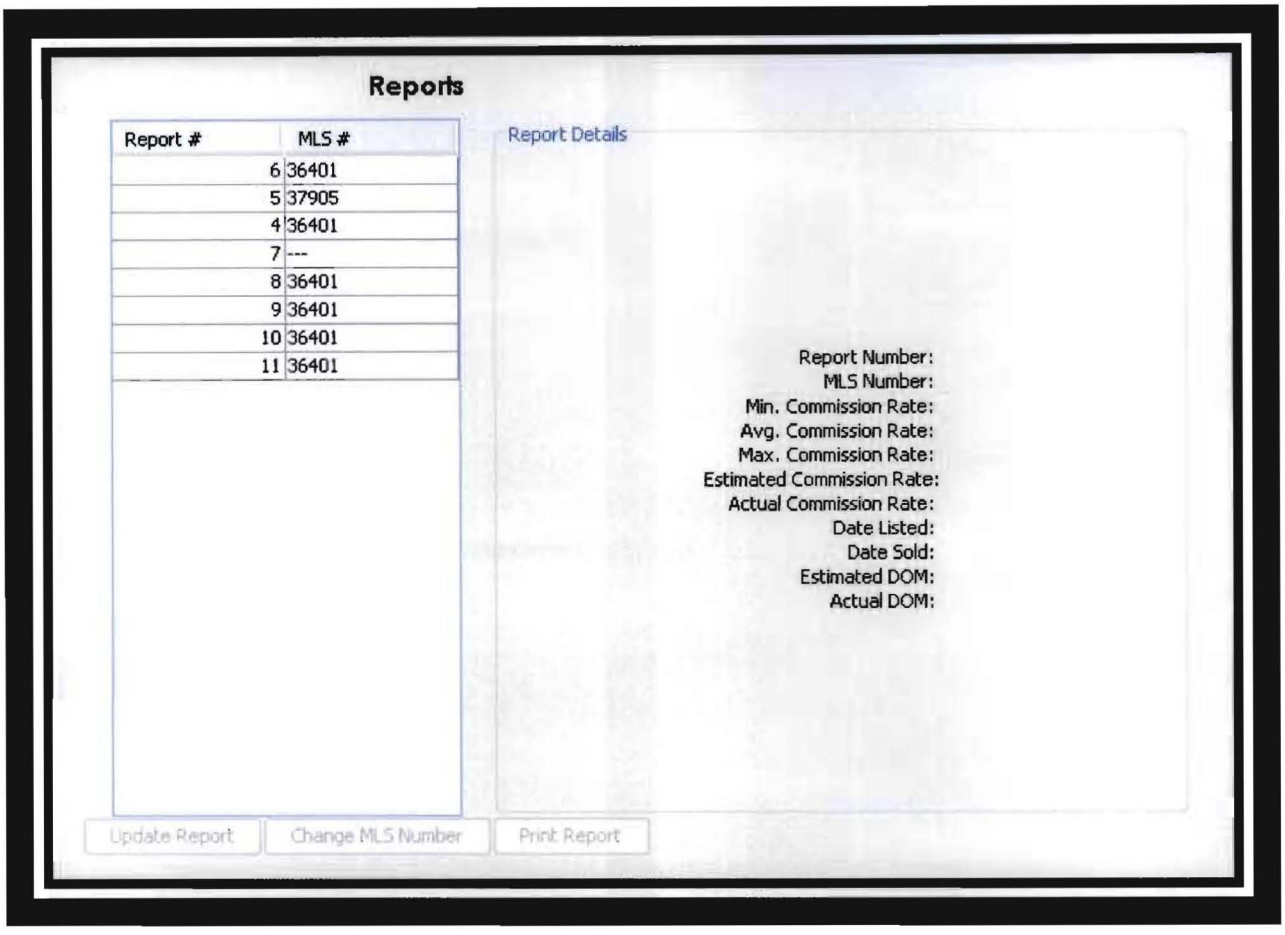
Testing this screen of our application became increasingly important to provide full condition coverage and path coverage in the core of our application. In order to achieve maximum efficiency, all testing was done with each field enabled. Then each individual field was disabled to check and see if it it worked standing alone. A random selection of combined fields were then tested to see how each field worked in tandem with another field, meaning every field was tested with at least one additional field enabled. With no fields enabled an error is produced.

Finally, a standard graph matrix was constructed to check invalid cases for each field and a fully valid test case was run.



	1	2	3	4	5	6	7	8	9
askP	V	I	v	v	v	v	v	v	v
askM	V	v	I	v	v	v	v	v	v
mAsk	V	v	v	I	v	v	v	v	v
sqrP	V	v	v	v	I	v	v	v	v
sqrM	V	v	v	v	v	I	v	v	v
mSqr	V	v	v	v	v	v	I	v	v
aceP	V	v	v	v	v	v	v	I	v
aceM	V	v	v	v	v	v	v	v	I
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
mAce	V	I	v	v	v	v	v	v	v
Zip	V	v	I	v	v	v	v	v	v
Town	V	v	v	I	v	v	v	v	v
Elem	V	v	v	v	I	v	v	v	v
Midd	V	v	v	v	v	I	v	v	v
High	V	v	v	v	v	v	I	v	v
PrimG	V	v	v	v	v	v	v	I	v
SecG	V	v	v	v	v	v	v	v	I
	1	**2**	**3**	**4**	**5**				
Found	V	I	v	v	v				
Bed	V	v	I	v	v				
Fb	V	v	v	I	v				
Hb	V	V	v	v	I				

askP: Asking Price Potential field
askM: Asking Price Maximum field
mAsk: Asking Price Minimum field
sqrP: Square Footage Potential field
sqrM: Square Footage Maximum field
mSqr: Square Footage Minimum field
aceP: Acreage Potential field
aceM: Acreage Maximum field
mAce: Acreage Minimum field
Zip: Zip Code field
Town: Township field
Elem: Elementary School field
Midd: Middle School field
High: High School field
PrimG: Primary Garage field
SecG: Secondary Garage field
Found: Foundation field
Bed: Bedrooms field
Fb: Full Baths field
Hb: Half Baths field



Update Report

CASE 1	CASE 2
Valid MLS Number	Invalid MLS Number

Change MLS Number

CASE 1	CASE 2
Valid MLS Number	Invalid MLS Number

An error can occur if you update the MLS number to an invalid value, either a non-integer or an integer that does not correspond to an MLS number in the RETS database. Printing does not cause an error within the program – any print errors will be handled by the operating system.

Summary

CommCalc is a program that calculates a consistently profitable commission rate for property listings. Currently, our client partner has neither software nor a defined set of procedures to accomplish this task. All commission rates are determined based on the experience and intuition of the agent listing the property. With the information that this system produces, realtors can also determine whether or not a listing should be accepted.

The goal of the project is to create software to calculate the best commission rate for a given listing based on historical data and fixed costs. When commission rates charged for various properties are totaled, they must cover all monthly expenses of running the business. In addition to using trends from historical data to determine the commission rate, variable costs such as utilities and payroll may be entered or updated by the client user to give a more accurate commission estimate.

In order to accomplish these goals, we have gone through a significant number of steps. We have come up with estimates for LOC (lines of code) using the Delphi Technique and effort using the COCOMO software provided to the class. We have designed our project to meet the requirements gathered from our client partner.

We designed our application with various OO analysis diagrams. These diagrams include a use-case diagram, a static structure (class) diagram, a sequence diagram, a high-level component diagram, and a deployment diagram. Along with these documents, we created a data dictionary to provide details that may not be evident just from looking at the visual form. These diagrams helped our team and our client partner to visualize the system and the structure to be used for implementation. Additionally, this exercise helped us to define key features of the program and how they interact.

As stated above, our client partner had no software or system in place to calculate the best commission rate for a given listing. Instead, listing agents followed some basic guidelines set forth by the company to arrive at a commission rate, including a minimum commission rate. The maximum commission rate was determined by the agent and market conditions. The agents reported to the administrative staff and management team at the office, and management had the final say in commission rates, but the agents were largely in control of the final value.

Given current market conditions, it is more important than ever that realty companies set the commission rate "just right" - too high and customers will list with other agencies, too low and the company will lose money on each sale. The current method is prone to human miscalculation and bias, traits that could be disastrous for any business.

Currently, there is no method for calculating a profitable commission rate. Any problems that have existed prior to the new system are the result of the limited capacity of human analysis and human error. These issues are present in the current state of our client partner, Century 21, because it is not possible for one person to take into account all available information. The actual amount of error remains unknown, however, because there is no way to measure the performance of the agents regarding optimal commission calculations. Therefore, a software system must be developed to handle a task that is currently handled by individual real estate agents.

It is important to note that while this system solves the aforementioned problems it is no substitute for the experience of a seasoned realtor. Rather, it is the intention of this program to provide guidance to the informed user. It is our hope that in the right hands this application will reap many years of profitable benefits for our client partner. This system will be especially important in a market that is currently unfavorable for real estate companies. It will give our client a competitive advantage over other businesses in the field. As with any business, if decision making is not constantly evolving and improving, the business will falter, fail, and disappear while its competitors grow and take over. Using computer software is one way such an improvement can be made.

Our final system proposal was a traditional client-server application. Users run *CommCalc* and use it to interact with two sets of data, a remote RETS database and a local MySQL database. A local client allows us to overcome the shortcomings of a web browser, and the centralized data servers still allow multiple users to connect to the system at any given time.

Use Cases

Use Case: Log In

Main Success Scenario:

1. User launches application.
2. Application displays login screen.
3. User enters username and password.
4. User clicks Submit or hits the Enter key.
5. System passes username and password to authentication subsystem.
6. Authentication system confirms credentials.
7. System initializes the user's session.

Extensions:

3. a. Authentication system denies credentials.
   b. System displays notification to user that the username or password is incorrect.

Use Case: Authentication

Main Success Scenario:

1. Authentication system receives username and password.
2. Authentication system queries database for a username and password match.
3. MySQL database replies with exactly one result.
4. Authentication system notifies calling system of success.

Extensions:

3. a. MySQL database replies with zero results.
   b. Authentication system notifies the calling system that authentication failed.

Use Case: Main Menu

Main Success Scenario:

1. System presents user with Main Menu welcome screen.
2. User selects an action.
3. System passes control to the selected subsystem.

Extensions:

3. a. User chooses the Exit command.
   b. Log Out function is called.
   c. Application exits and returns control to the operating system.

Use Case: Log Out

Main Success Scenario:

1. Log Out function is called.
2. Application destroys the user's session.
3. Returns to the Log In screen.

Use Case: Exit

Main Success Scenario:

1. Log Out function is called and session is destroyed if user is logged in.
2. The application is closed.

Use Case: Add Fixed Cost Type

Main Success Scenario:

1. User selects Add Fixed Cost from Fixed Cost under the Tools menu.
2. Application displays Add Fixed Cost Type Screen.
3. User types a brief description of the fixed cost type to be added.
4. User clicks Submit or hits the Enter key.
5. System passes description to the Data Commit subsystem.
6. Data Commit subsystem notifies Add Fixed Cost Type subsystem that data has been saved successfully.

Extensions:

6. a. Data Commit subsystem notifies Add Fixed Cost Type subsystem that the description already exists.
   b. System displays notification to the user.

Use Case: Add Fixed Cost

Main Success Scenario:

1. User selects Manage Fixed Cost from Fixed Cost under the Tools menu.
2. Manage Fixed Costs subsystem submits a request to the Data Search subsystem for the

current fixed costs.
3. Application displays Manage Fixed Cost Screen with scrollable, sortable list of costs.
4. User clicks Add button.
5. A dialog box with four fields appears.
6. User selects a fixed cost type from the drop down menu.
7. User enters an amount for the fixed cost.
8. User selects a month and year for the cost.
9. User identifies if this entry is an income or an expense.
10. User clicks Add button or presses the Enter key.
11. System passes the fixed cost data to the Data Commit subsystem.
12. Data Commit subsystem notifies Manage Fixed Cost subsystem that data has been saved successfully.
13. Manage Fixed Costs subsystem returns to its entry point with all of the current fixed cost values.

Extensions:

10. a. User clicks Cancel button in the dialog box.
    b. Dialog box closes without making any new additions.
    c. Returns to Manage Fixed Cost Screen.
12. a. Data Commit subsystem notifies Manage Fixed Cost subsystem that the fixed cost already exists.
    b. System displays notification to the user.

Use Case: Edit Fixed Cost

Main Success Scenario:

1. User selects Manage Fixed Cost from Fixed Cost under the Tools menu.
2. Manage Fixed Costs subsystem submits a request to the Data Search subsystem for the current fixed costs.
3. Application displays Manage Fixed Cost Screen with scrollable, sortable list of costs.
4. User selects a fixed cost to be edited by clicking and highlighting a row.
5. User clicks the Edit button.
6. System presents a dialog box with the selected fixed cost information displayed.
7. User modifies the existing data in the fields.
8. User clicks the Edit button or presses the Enter key.
9. System passes the input to the Data Commit subsystem.
10. Data Commit subsystem notifies the Manage Fixed Costs subsystem that the data has been saved successfully.
11. Manage Fixed Costs subsystem returns to its entry point with all of the current fixed cost values.

Extensions:

8. a. User clicks Cancel button in the dialog box.
   b. Dialog box closes without making any changes.
   c. Returns to Manage Fixed Cost Screen.
10. a. Data Commit subsystem notifies Manage Fixed Cost subsystem that the fixed cost already exists.
    b. System displays notification to the user.

Use Case: Delete Fixed Cost

Main Success Scenario:

1. User selects Manage Fixed Cost from Fixed Cost under the Tools menu.
2. Manage Fixed Costs subsystem submits a request to the Data Search subsystem for the current fixed costs.
3. Application displays Manage Fixed Cost Screen with scrollable, sortable list of costs.
4. User selects a fixed cost to be deleted by clicking and highlighting a row.
5. User clicks the Delete button.
6. System passes the fixed cost data to the Data Commit subsystem.
7. Data Commit subsystem notifies the Manage Fixed Costs subsystem that the data has been deleted.
8. Manage Fixed Costs subsystem returns to its entry point.

Use Case: Calculate Commission Rate

Main Success Scenario:

1. User selects Calculate Commission Rate from the Tools menu.
2. Application displays Calculate Commission Rate Screen with search criteria fields.
3. User checks boxes in the Use column if he or she wishes to use this criteria in the search for similar listings.
4. User fills in data for the potential listing that a commission rate is being calculated for.
5. Data Search subsystem provides configuration values used to pre-populate range values.
6. User may modify the range values for a broader or narrower search.
7. User clicks the Calculate button.
8. System passes the input to the Data Search subsystem.
9. Data Search subsystem returns information for similar listings.
10. System displays a calculated commission rate, an estimated number of Days on Market (DOM), and other information regarding similar listings.
11. User clicks OK when finished.
12. Application returns to the Calculate Commission Rate Screen.

Extensions:

8. a. An error is displayed to the user if a field is selected for use and left empty, if

characters are entered where numerical values are expected, or if there is no search criteria provided.
b. User clicks OK.
c. Application returns to the Calculate Commission Rate Screen.

9. a. Data Search subsystem notifies Calculate Commission Rate subsystem that the search criteria was too restrictive.
b. System displays notification to the user.

Use Case: Configure

Main Success Scenario:

1. User selects Configure from the Admin menu.
2. Configure subsystem submits request to Data Search subsystem for the current configuration values.
3. Application displays Configure Screen with a list of configuration options.
4. User may modify the configuration values to change the default behavior of the program.
5. User clicks the Update button.
6. System passes the input to the Data Commit subsystem.
7. Data Commit subsystem notifies the Manage Fixed Costs subsystem that the data has been saved successfully.

Extensions:

7. a. Data Commit subsystem notifies Configure subsystem that configuration values failed to be updated.
b. System displays notification to the user.

Use Case: View Logs

Main Success Scenario:

1. User selects View Logs from the Admin menu.
2. View Logs subsystem submits a request to the Data Search subsystem for the Created Users logs.
3. Application displays View Logs Screen with logs for Created Users as the initial display.
4. User may change the logs displayed by clicking on Created Users, Created Fixed Costs, Updated Users, Updated Fixed Costs, or Deleted Fixed Costs.
5. View Logs subsystem submits a request to the Data Search subsystem for the logs selected by the user.
6. System is refreshed and the new logs are displayed.
7. User may click Refresh to update data in the current logs.

Use Case: Update Report MLS Number

Main Success Scenario:

1. User selects View Reports from the Tools menu.
2. View Reports subsystem submits a request to the Data Search subsystem for all available reports.
3. Application displays View Reports Screen with a list of reports on the left and a display area on the right.
4. User selects a report from the list by clicking and highlighting the row.
5. User clicks the Change MLS Number button.
6. System presents a dialog box where the MLS number can be added.
7. User enters the MLS number associated with the property in the report.
8. User clicks the Add button.
9. System passes the input to the Data Commit subsystem.
10. Data Commit subsystem notifies the View Reports subsystem that the data has been saved successfully.
11. View Reports subsystem returns to its entry point.

Extensions:

8. a. User clicks Cancel button in the dialog box.
   b. Dialog box closes without making any new additions.
   c. Returns to View Reports Screen.
10. a. Data Commit subsystem notifies View Reports subsystem that the MLS number failed to be saved.
    b. System displays notification to the user.

Use Case: Update Report

Main Success Scenario:

1. User selects View Reports from the Tools menu.
2. View Reports subsystem submits a request to the Data Search subsystem for all available reports.
3. Application displays View Reports Screen with a list of reports on the left and a display area on the right.
4. User selects a report from the list by clicking and highlighting the row.
5. User clicks the Update button.
6. System queries the Data Search subsystem for updated listing values.
7. Updated values are retrieved and stored in the local database.
8. Visible information is refreshed on screen.
9. View Reports subsystem returns to its entry point.

Extensions:

6. a. MLS number has not yet been associated with this report and updated information cannot be retrieved from the RETS database.
   b. System displays notification to the user.

Use Case: Print Report

Main Success Scenario:

1. User selects View Reports from the Tools menu.
2. View Reports subsystem submits a request to the Data Search subsystem for all available reports.
3. Application displays View Reports Screen with a list of reports on the left and a display area on the right.
4. User selects a report from the list by clicking and highlighting the row.
5. User clicks the Print button.
6. System submits the report to the operating system for printing.
7. User follows print instructions provided by the operating system.
8. View Reports subsystem returns to its entry point.

Use Case: Add User

Main Success Scenario:

1. User selects Manage Users from the Admin menu.
2. Manage Users subsystem submits a request to the Data Search subsystem for the current user accounts.
3. Application displays Manage Users Screen with scrollable, sortable list of users.
4. User clicks the Add button.
5. System presents a dialog box with four fields.
6. User enters Username and Password.
7. User selects Privilege Level from the drop down menu.
8. User enters the real name of the user in the Name field.
9. User clicks the Add button or presses the Enter key.
10. System passes the input to the Data Commit subsystem.
11. Data Commit subsystem notifies the Manage Users subsystem that the data has been saved successfully.
12. Manage Users subsystem returns to its entry point with all current users displayed.

Extensions:

9. a. User clicks Cancel button in the dialog box.
   b. Dialog box closes without making any new additions.

c. Returns to Manage Fixed Cost Screen.

11. a. Data Commit subsystem notifies Manage Users subsystem that the user already exists.
    b. System displays notification to the user.

Use Case: Edit User

Main Success Scenario:

1. User selects Manage Users from the Admin menu.
2. Manage Users subsystem submits a request to the Data Search subsystem for the current user accounts.
3. Application displays Manage Users Screen with scrollable, sortable list of users.
4. User selects a user to be edited by clicking and highlighting a row.
5. User clicks the Edit button.
6. System presents a dialog box with the selected user information displayed.
7. User modifies the existing data in the fields.
8. User clicks the Edit button or presses the Enter key.
9. System passes the input to the Data Commit subsystem.
10. Data Commit subsystem notifies the Manage Users subsystem that the data has been saved successfully.
11. Manage Users subsystem returns to its entry point with all current users displayed.

Extensions:

8. a. User clicks Cancel button in the dialog box.
   b. Dialog box closes without making any changes.
   c. Returns to Manage Users Screen.
10. a. Data Commit subsystem notifies Manage Users subsystem that the user already exists.
    b. System displays notification to the user.

Use Case: Set User Password

Main Success Scenario:

1. User selects Manage Users from the Admin menu.
2. Manage Users subsystem submits a request to the Data Search subsystem for the current user accounts.
3. Application displays Manage Users Screen with scrollable, sortable list of users.
4. User selects a user by clicking and highlighting a row.
5. User clicks the Set Password button.
6. System presents a dialog box two fields for setting and confirming the new password.
7. User enters the new password twice.
8. User clicks the Edit button or presses the Enter key.

9. System passes the input to the Data Commit subsystem.
10. Data Commit subsystem notifies the Manage Users subsystem that the data has been saved successfully.
11. Manage Users subsystem returns to its entry point with all current users displayed.

Extensions:

8. a. User clicks Cancel button in the dialog box.
   b. Dialog box closes without making any changes.
   c. Returns to Manage Users Screen.
10. a. Data Commit subsystem notifies Manage Users subsystem that the two passwords entered do not match and cannot be committed.
    b. System displays notification to the user.

Use Case: Enable/Disable User

Main Success Scenario:

1. User selects Manage Users from the Admin menu.
2. Manage Users subsystem submits a request to the Data Search subsystem for the current user accounts.
3. Application displays Manage Users Screen with scrollable, sortable list of users.
4. User checks the box in the Enabled column to enable a user or deselects the box to disable the user.
5. System passes the input to the Data Commit subsystem.

Use Case: Data Commit

Main Success Scenario:

1. System passes user input to Data Commit subsystem.
2. This system confirms that data is in the correct format.
3. System arranges and prepares data for database insertion.
4. System passes data to the database.
5. This subsystem passes control back to the calling system.

Extensions:

2. a. System discovers missing or malformed input.
   b. System notifies user of the error.
   c. System passes control back to the calling system, along with a notification that an error occurred.

Use Case: Data Search

Main Success Scenario:

1. User search query data is passed to Data Search subsystem.
2. System prepares user input for submission to database.
3. System submits data to database.
4. Database replies with query results.
5. System formats results into results object.
6. Results object is returned to the calling subsystem.

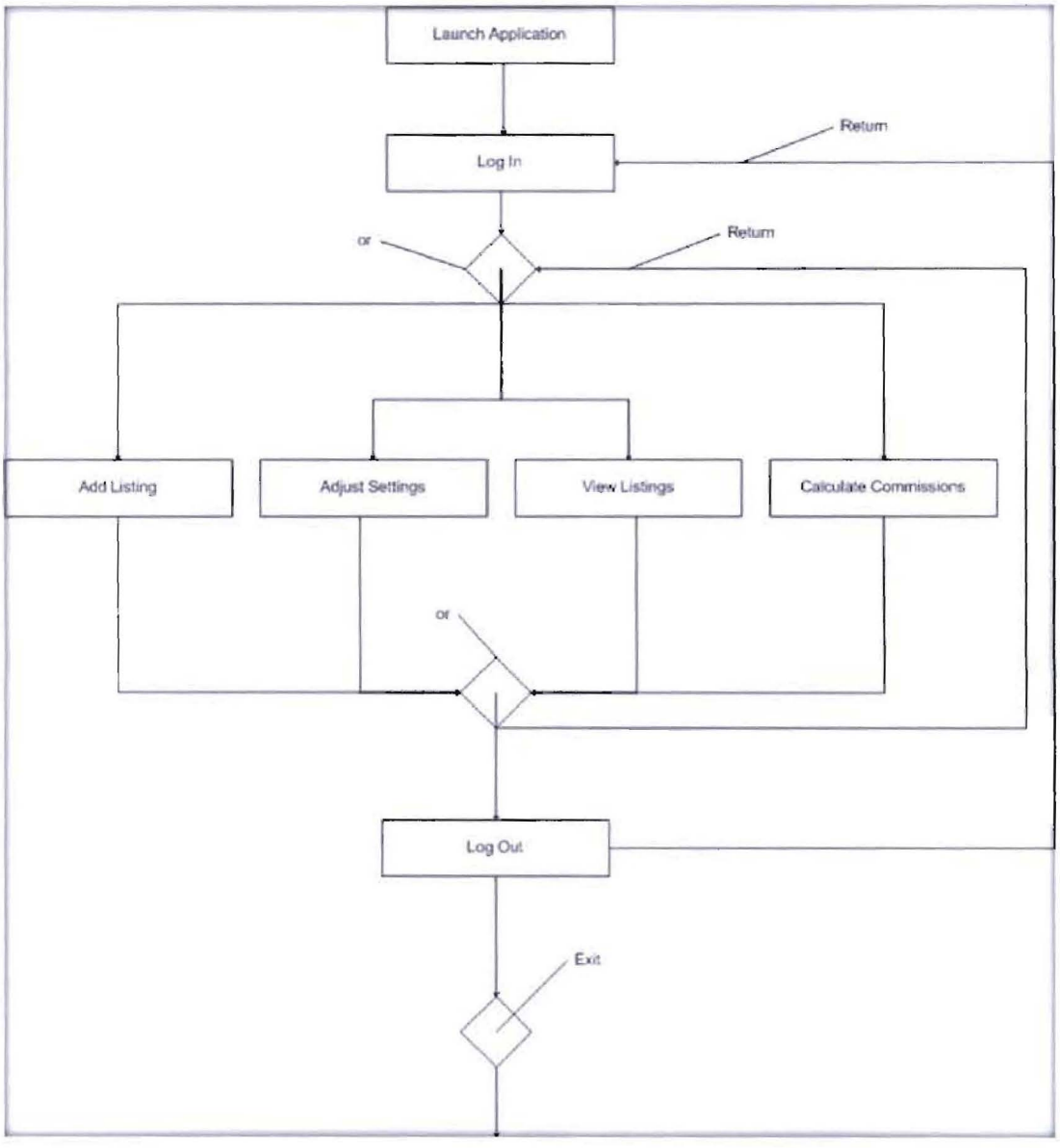
Extensions:

4. a. If no results are returned, a notification will be displayed to the user.

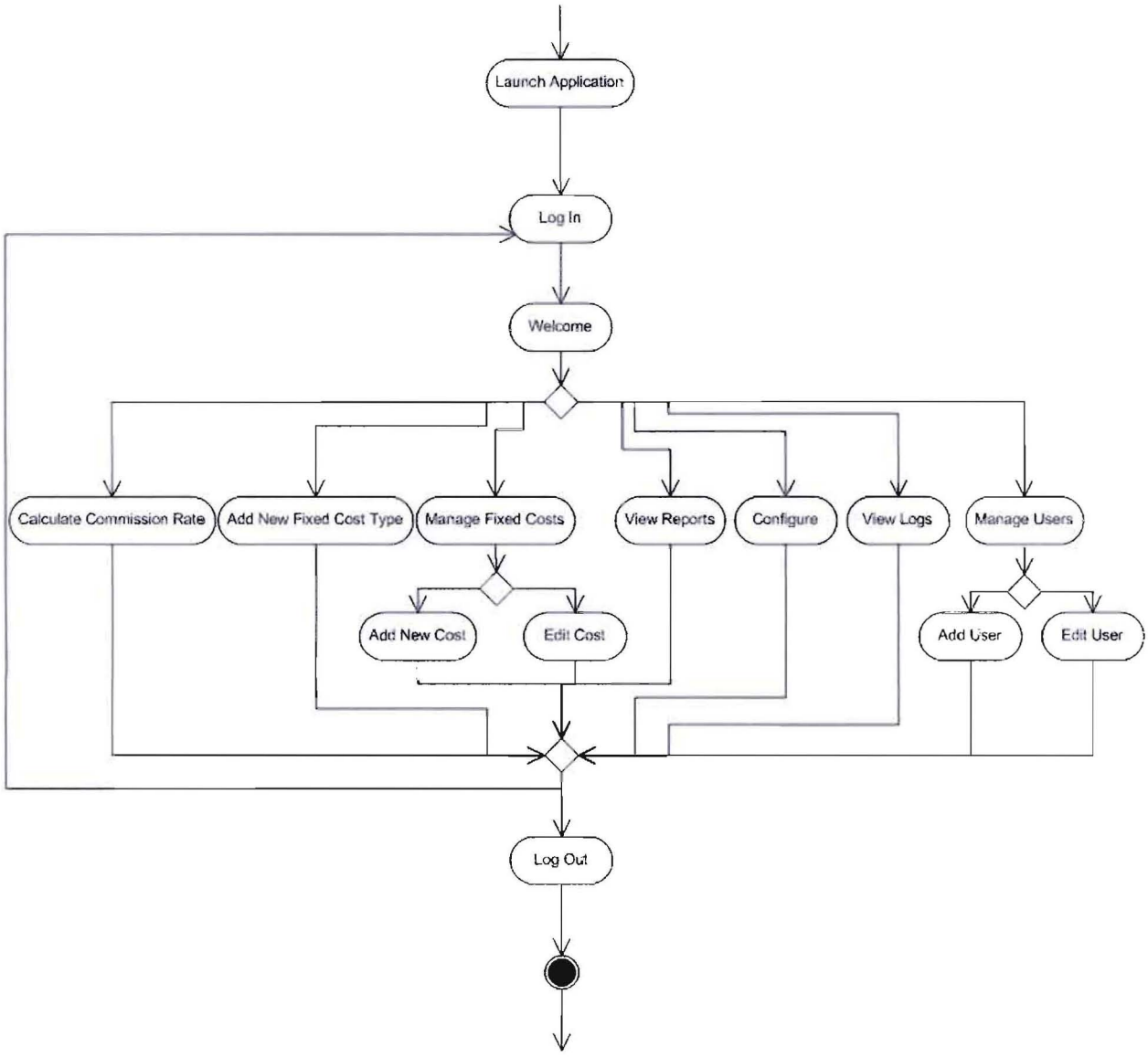
Design History

The figure on page 32 shows our current module breakdown, but we weren't able to achieve this type of granularity overnight. Our design documents evolved throughout the year as we collaborated with our client-partner, clarified our requirements, and gained a better understanding of the system we were developing. These documents helped to guide our programming and allowed us to document the changes that we found necessary. Being an important part of the development process, the entire design history is archived in this section. Provided below are the various diagrams that we used as well as explanations for each step.

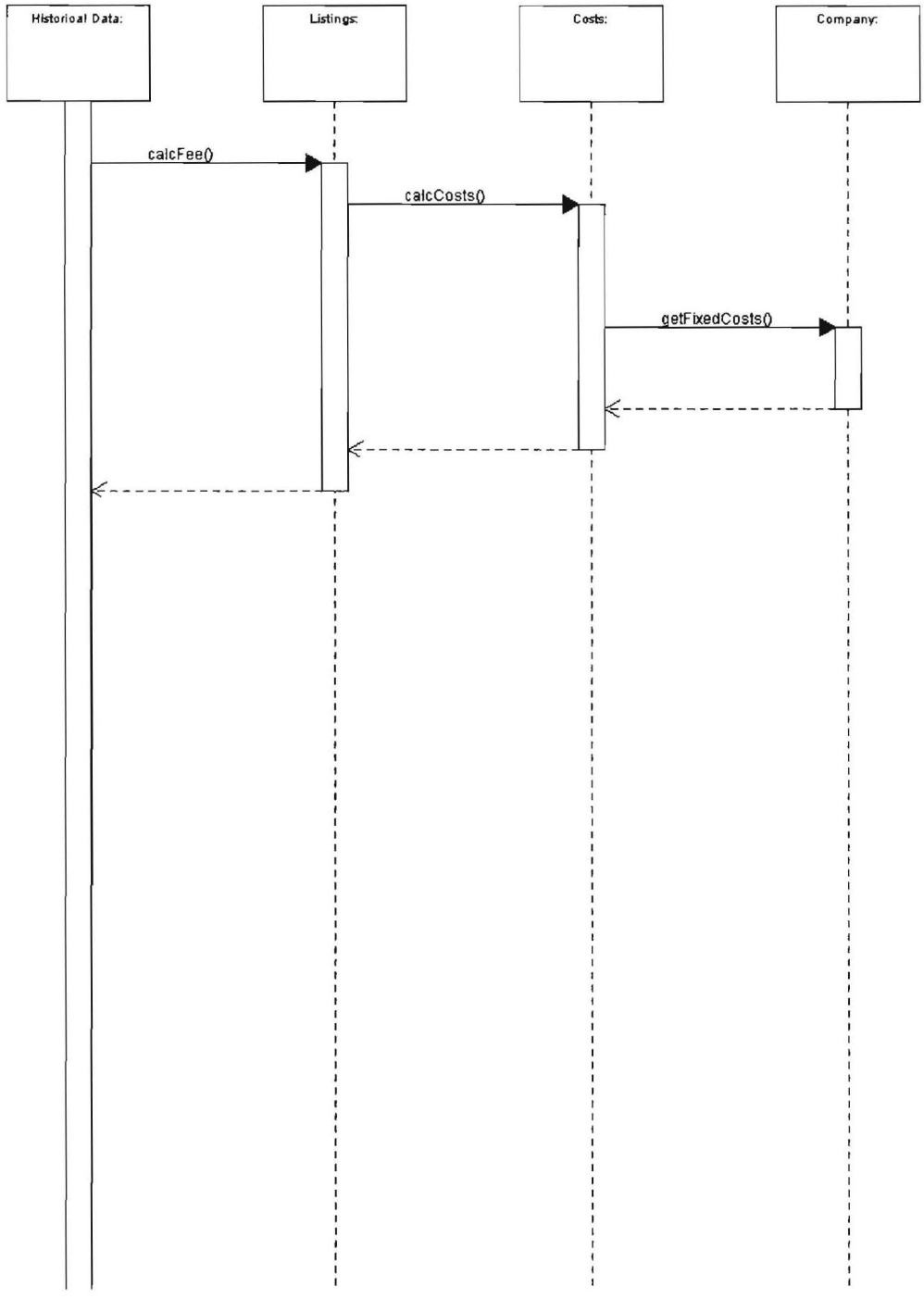
A good starting point is the high-level component diagram. This diagram was rather vague and helped our team to get a view of the system as a whole and the functions it needed to perform. Our original high-level component diagram is shown at the top of the next page.



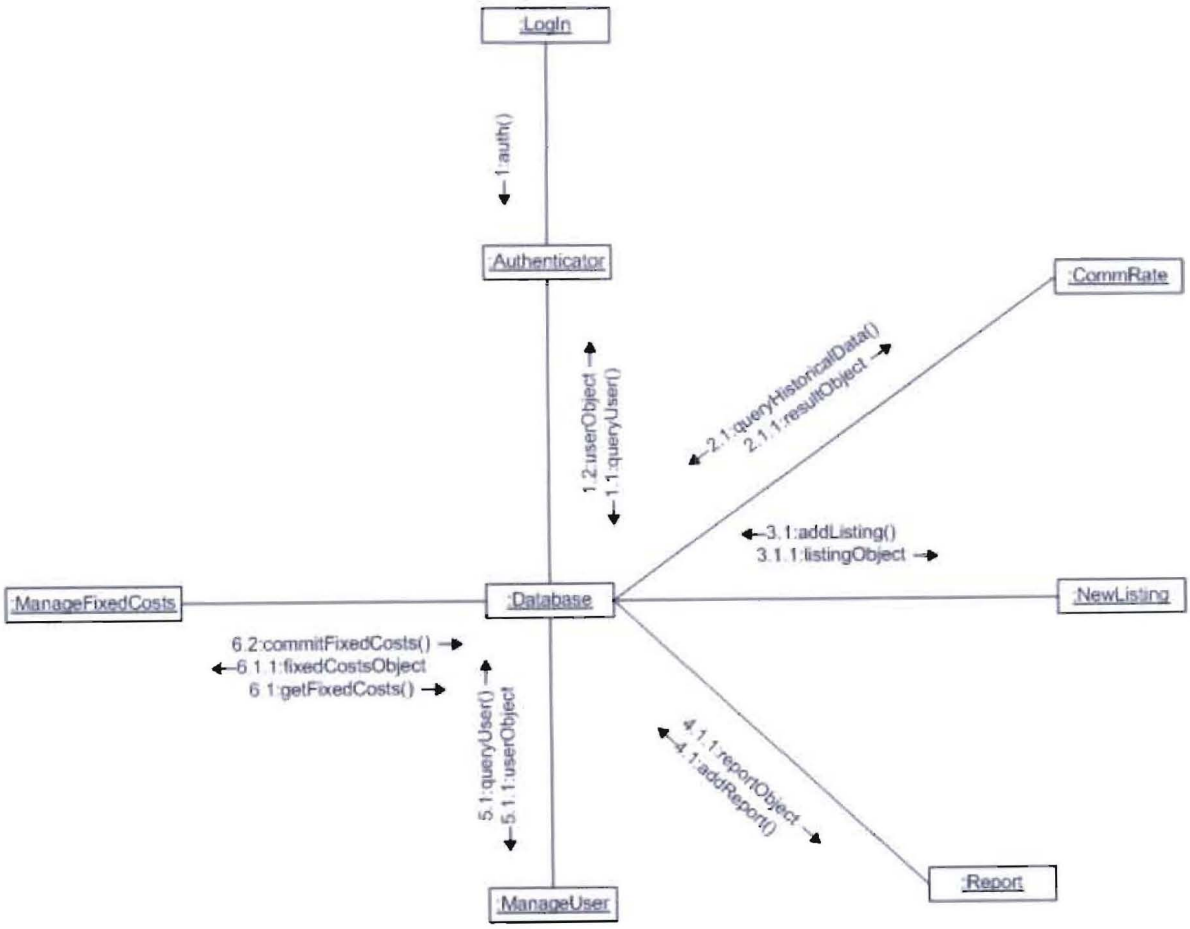
Even this diagram, general as it was, underwent some changes. As our work progressed, the design became more complex and definite. This can be seen in our final high-level component diagram in the next illustration.



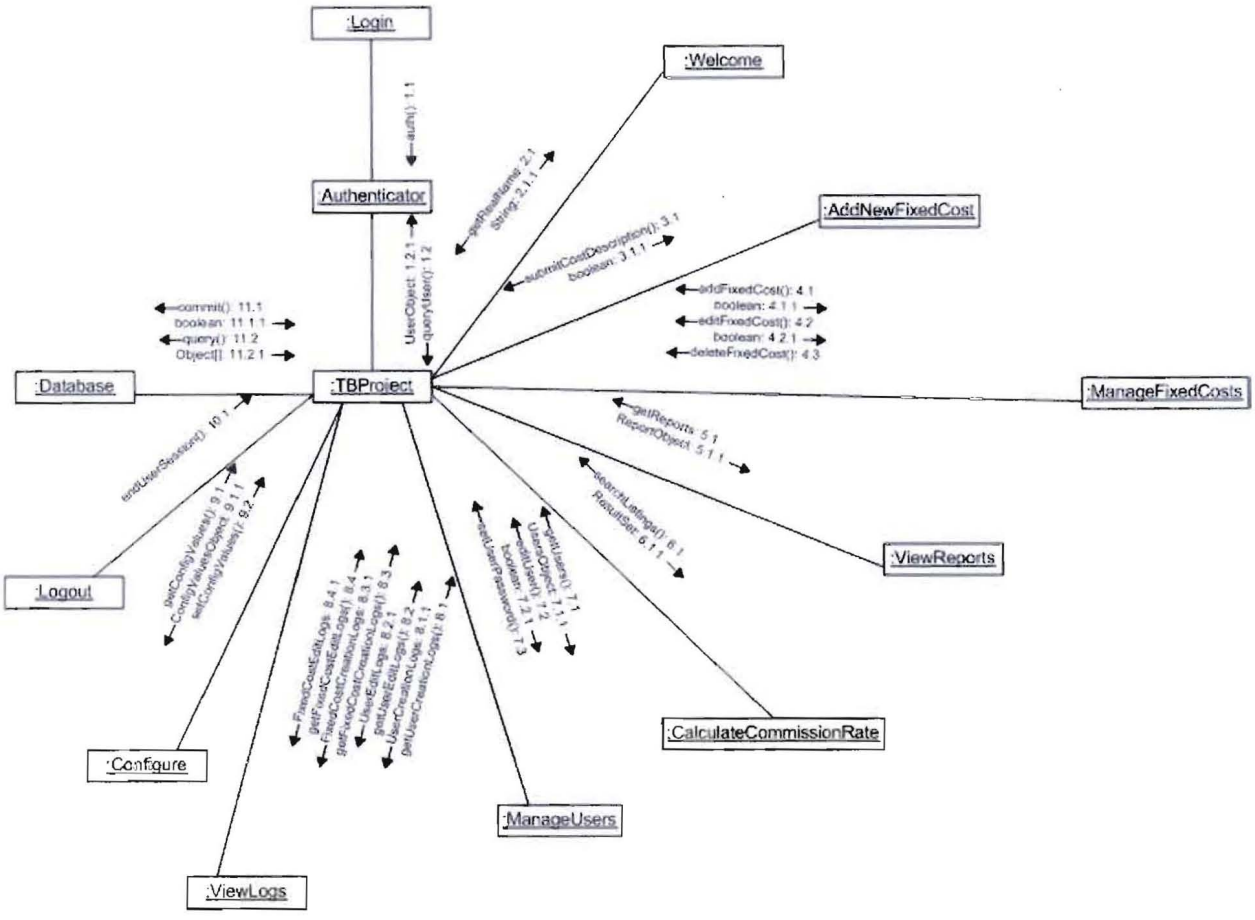
Another of our early diagrams was the sequence diagram. This is intended to demonstrate a series of events that would occur during the use of the application. Process control and time are key elements of this system representation. The sequence diagram is displayed on the next page.


Historical Data:
Listings:
Costs:
Company:
calcFee()
calcCosts()
getFixedCosts()

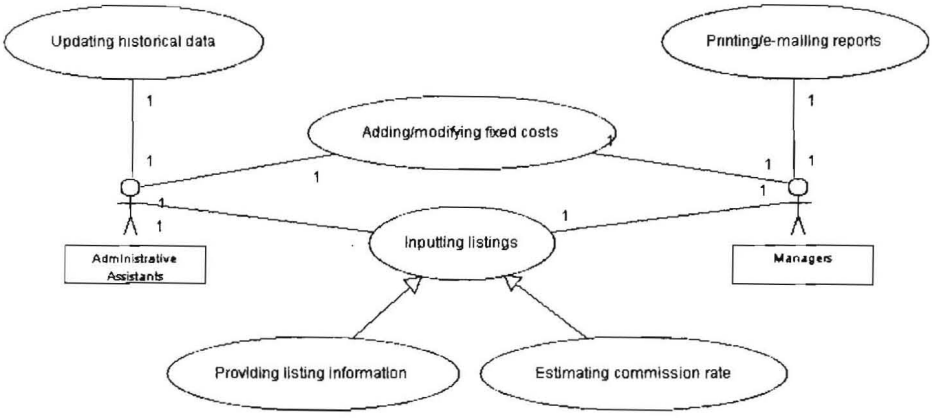
After the design reviews with industry professionals, we decided that the sequence diagram was not an appropriate design tool for our system because timing was in no way critical to the functionality of our application. At the suggestion of one of our reviewers, we turned to the use of collaboration diagrams instead. This diagram still shows all of the functionality and relationships but without the unnecessary element of time. Our initial collaboration diagram can be seen below:



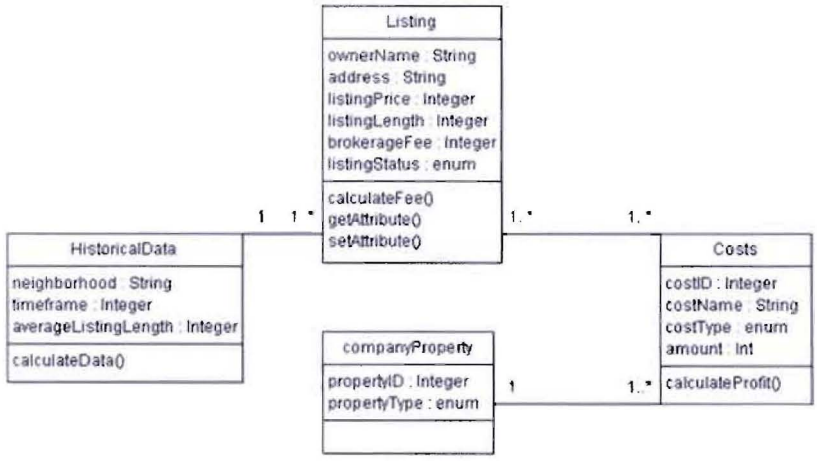
This diagram was also changed and refined over time. This particular diagram became helpful during the testing process and can be seen earlier in the document. As an important part of the design history, the diagram is reproduced on the next page.



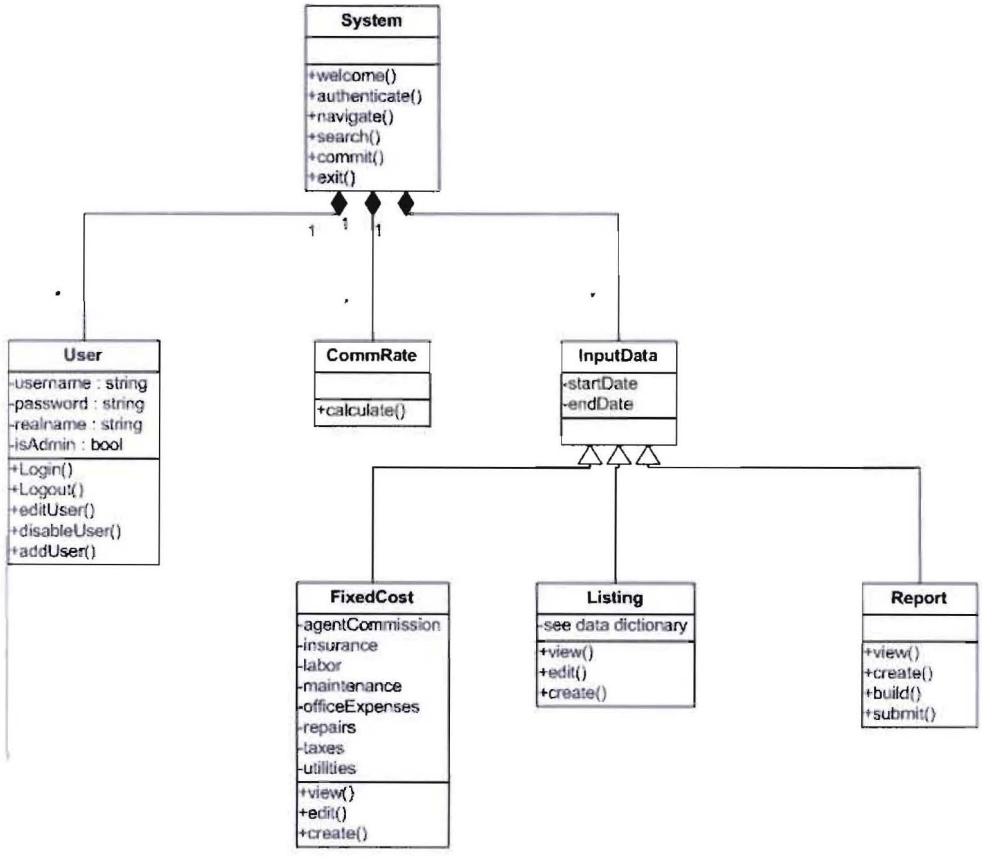
The use case diagram, shown below, was another document in the design process. This was a general representation of the user roles for the application. This diagram did not undergo intense development because it had limited implications for the system and there were not many different users to consider. This was still helpful in communicating with our client-partner since it provided a visualization of the users and tasks to be performed.



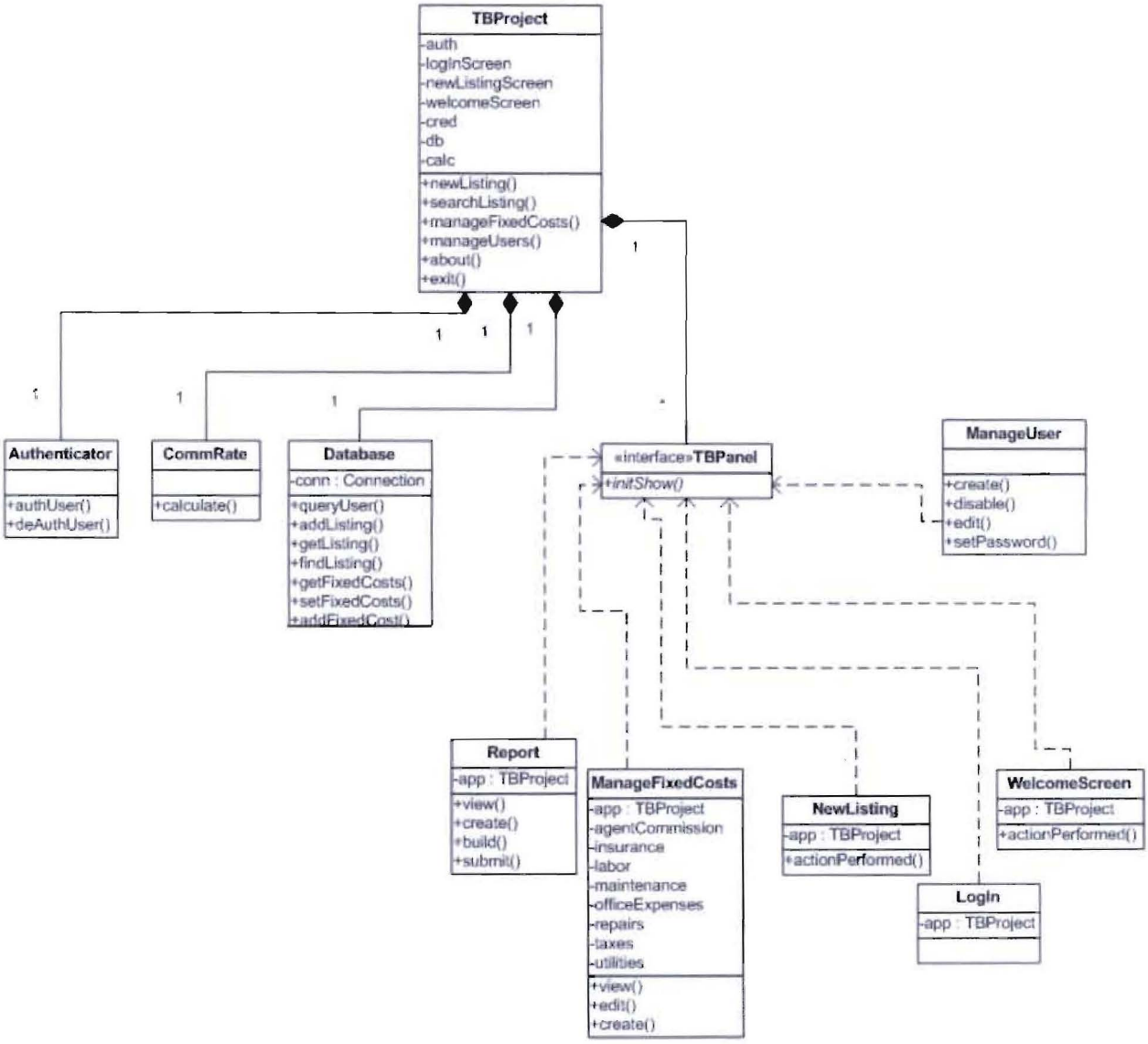
Finally, we developed a series of class diagrams. These were probably the most important and helpful to the development process. These diagrams showed a much lower level of detail and guided our work. Our earliest class diagram still left much to be desired though:



The first revision to this diagram established the first real module structure for the application. This was an important step. Adding structure helped to show relationships, organize classes, and create packages that would represent different groups of related modules.



As we began implementing the code, we saw a few things that needed to be revised in our design documents. For instance, we realized we had more screens and functions than we had originally thought. We also found that the use of InputData as a generalization of various user inputs was superfluous. We also found the need to add an Authenticator class for handling user login.



Although we were getting closer, we were still not finished with the class diagram development. As shown earlier in the manual, the final class diagram was much more detailed and complex. In the end, the connections were rather more difficult to follow, but the operations listed for each class covered all of the program functionality.